Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

ATTACCHI MICROARCHITETTURALI

Relatore:
Chiar.mo Prof.
Ozalp Babaoglu

Presentata da:
Simone Fabbri

Sessione III
Anno Accademico 2018 - 2019

Being a person can be noble, but not shameless; being a person can be not great, but not despicable.

Introduzione

Gli attacchi microarchitetturali sono una tipologia di attacchi che non sfruttano, come da consuetudine, difetti nel software. Sfruttano bensì difetti nella microarchitettura dei moderni elaboratori rendendoli potenzialmente molto pericolosi, essendo (in linea teorica) indipendenti dal sistema operativo installato, da particolari versioni di specifici programmi e perfino in grado di agire in ambienti "sandbox" e macchine virtuali. Con microarchitettura possiamo intendere l'implementazione dell'ISA (Instruction Set Architecture), una sorta di interfaccia tra l'hardware e il software che rappresenta un modello astratto di un computer (x86, ARMv8, SPARC, ...) [1].

Gli attacchi microarchitetturali che saranno in seguito illustrati sono una tipologia dei cosidetti **side-channel attacks**, che sfruttano meta-informazioni (informazioni sulle informazioni). Per fare un'analogia, si pensi allo scassinatore che con uno stetoscopio ascolta i rumori prodotti da una cassaforte mentre gira la manopola, per scoprire la combinazione segreta. Questo è possibile perché elementi microarchitetturali dell'hardware lasciano tracce durante l'esecuzione di un comando, e possono rivelare altre informazioni (e.g. timing) che è possibile sfruttare per carpire dati sensibili, a cui l'utente che le ha generate normalmente non vuole che si acceda. Gli attacchi considerati sono inoltre basati in software, ovvero non occorre accedere fisicamente all'hardware per eseguirli.

La tesi è così composta: nella prima parte si introdurranno alcune conoscenze fondamentali su specifici elementi dell'architettura dei moderni elaboratori. Successivamente, si cercherà di dare una panoramica su questi

attacchi, con più riferimenti possibili allo stato dell'arte attuale. Si introdurranno anche delle tecniche attualmente in uso per difendersi da questi attacchi; si proverà infine a rispondere ad alcune domande sulla pericolosità di questi attacchi: se rappresentano una vera minaccia per la sicurezza pubblica, chi è più a rischio (server esposti tutto il giorno, tutti i giorni oppure utenti privati) e chi è in grado di eseguire questi attacchi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Background

In questo capitolo si forniranno elementi di background che saranno utili per discutere successivamente nel dettaglio il funzionamento degli attacchi microarchitetturali basati su software; si darà quindi una semplice spiegazione di come è organizzata una CPU e di come si possa ottimizzare la velocità di esecuzione delle istruzioni, cos'è e come funziona la memoria cache nelle CPU, cosa sono la branch prediction e la speculative execution, come funziona la memoria virtuale e la DRAM.

1.1 Central Processing Unit

La CPU (**C**entral **P**rocessing **U**nit) è la parte di un elaboratore che si occupa di svolgere i calcoli. È formata da più parti distinte; infatti dispone di una unità di controllo, in grado di prelevare e decodificare le istruzioni da eseguire, e una ALU (unità aritmetico logica) che esegue effettivamente le istruzioni. Una CPU dispone anche di numerosi registri, in cui è in grado di salvare informazioni utili alle elaborazioni che deve svolgere come ad esempio il **Program Counter**, il quale contiene la successiva istruzione che dovrà essere prelevata per l'esecuzione e l'**Instruction register**, che invece contiene l'istruzione in fase di esecuzione.

Grazie a questi elementi, la CPU è in grado di compiere il cosidetto *ciclo FDE* (fetch - decode - execute), che appunto è un'insieme di operazioni che si ripetono ciclicamente e consistono in:

1. Fetch dell'istruzione (caricamento nella cpu);
2. Decode dell'istruzione (decodifica dell'istruzione, interpretazione dell'istruzione);
3. Esecuzione dell'istruzione.

Un primo modo naive quindi per velocizzare l'esecuzione delle istruzioni è costruire CPU più potenti, in grado di eseguire il sopracitato ciclo FDE molto velocemente. Purtroppo ci sono dei limiti su che cosa si può ottenere mediante la semplice forza bruta; ci si è quindi concentrati anche su altri "trucchi" per ottimizzare la velocità delle CPU. Invece di concentrarsi sulla semplice velocità di esecuzione, è possibile lavorare su *processori multicore*, ovvero processori che hanno più componenti in grado di eseguire istruzioni. Ognuno dei core di un processore multicore ha alcune risorse private (e.g. registri) e alcune risorse condivise.

Altre soluzioni trattate nello specifico in questa parte sono le *memorie cache*, i sistemi di *branch prediction* e di *speculative execution*. Un fattore comune di questi tre elementi è il seguente: uno dei maggiori colli di bottiglia per il ciclo FDE è proprio il fetch delle istruzioni, dalla lenta memoria centrale alla CPU. Di conseguenza, queste tre soluzioni cercano di mitigare il problema, provando a indovinare i dati in anticipo o cercando di mantenerne una parte in memorie più veloci della memoria centrale.

1.2 Memoria Cache

Come citato sopra, le CPU sono molto più veloci della memoria centrale: il che significa che quando la CPU richiede un dato in memoria, potrebbe dover attendere molti cicli prima di ottenerlo. Anche se sarebbe possibile

installare memorie più veloci, questo comporterebbe costi molto elevati. Si è scelto quindi di mantenere delle grandi memorie più lente, e usare invece delle piccole memorie (la dimensione limitata dipende appunto dei costi) molto veloci, installate direttamente sulla CPU, chiamate memorie cache.

L'idea di base è molto semplice: i dati usati più di frequente da una CPU sono mantenute nella memoria cache, diminuendo quindi considerevolmente i tempi di accesso a quei dati.

In una moderna CPU sono presenti tipicamente 3 livelli di memoria cache, L1, L2 e L3, che vanno dalla meno capiente e più veloce (L1), alla più capiente e meno veloce (L3).

La memoria centrale viene suddivisa in blocchi di dimensione fissa chiamati **linee di cache**. Una linea di cache è composta generalmente da 4 a 64 byte consecutivi. Le linee sono numerate consecutivamente a partire da 0: la linea 0 conterrà i byte compresi tra 0 e 31, la linea 2 tra 32 e 63, e così via. In ogni momento la cache contiene delle linee; quando al processore occorre una parola in memoria, verifica se essa sia presente nella cache: se così non fosse, si elimina una linea dalla cache per fare posto ad una nuova linea proveniente dalla memoria centrale con dentro la parola richiesta, altrimenti si passano i dati al processore.

Ci sono diversi modi per organizzare una cache, ma ogni livello (L1, ..., Ln) utilizza lo stesso modello.

1.2.1 Cache a indirizzamento diretto

La forma più semplice di una cache è una cache a indirizzamento diretto (*direct-mapped cache*), in figura 1.1. Gli elementi della cache sono composti da 3 parti:

1. il bit `Valid` che indica se il dato nell'elemento è valido oppure no. All'avvio del sistema ogni elemento è marcato come non valido.

2. Il campo `Tag` che è un valore univoco a 16 bit, corrispondente alla linea di memoria da cui provengono i dati.

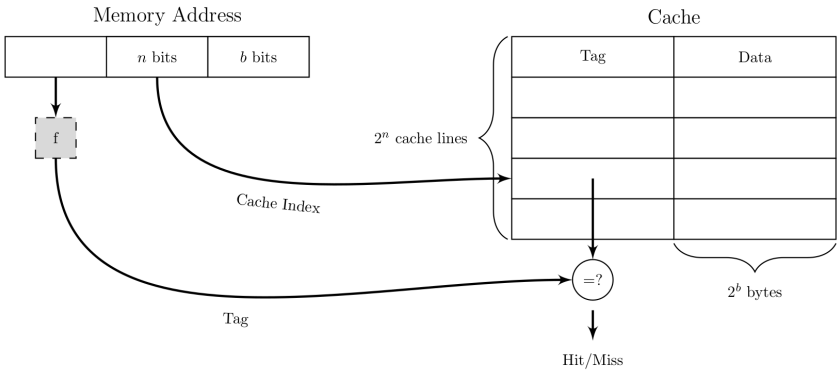

Figura 1.1: cache a indirizzamento diretto.

3. Il campo `Data` che contiene una copia del dato della memoria. Questo campo memorizza una linea di cache da e.g. 32 byte.

In questo modello di cache una data parola di memoria può essere memorizzata in un'unica posizione della cache. Per ogni indirizzo di memoria esiste un solo posto all'interno della cache in cui cercare il dato. Per memorizzare e prelevare dati dalla cache l'indirizzo è diviso a sua volta in 4 campi:

1. il campo `TAG` corrisponde ai bit "tag" memorizzati in un elemento della cache;

2. il campo `LINE` indica l'elemento della cache contenente i dati corrispondenti, se sono presenti;

3. il campo `WORD` indica a quale parola si fa riferimento all'interno della linea;

4. il campo `BYTE` non viene generalmente utilizzato, ma se si richiede un solo byte esso indica quale byte è richiesto all'interno della parola. Per una cache che fornisce soltanto parole a 32 bit, questo campo varrà sempre 0.

Quando la CPU genera un indirizzo di memoria, l'hardware estrae dall'indirizzo gli n bit del campo `LINE` (supponendo una cache a 2^n linee) e li

usa come indice all'interno della cache per cercare tra tutti gli elementi. Se questo elemento è valido si effettua un confronto tra il campo `TAG` dell'indirizzo di memoria e il campo `TAG` dell'elemento della cache. Se sono uguali, l'elemento della cache contiene la parola richiesta, e si parla di **cache hit**. In tal caso la parola che si vuole leggere può essere presa direttamente dalla cache. Dalla linea di cache viene estratta la parola effettivamente richiesta, mentre non viene utilizzato il resto della linea. Se l'elemento della cache non è valido oppure i due dati non corrispondono, il dato richiesto non è presente nella cache. Questa condizione viene detta **cache miss**. In questo caso la linea della cache a 32 byte viene prelevata dalla memoria e memorizzata nell'elemento appropriato della cache, sostituendo quello che era presente precedentemente.

Nonostante la complessità della decisione, l'accesso alla parola può essere notevolmente veloce. Nel momento in cui si conosce l'indirizzo, si conosce anche l'esatta locazione della parola (ammesso che si trovi nella cache). Ciò significa che è possibile leggere la parola e spedirla al processore nello stesso momento in cui esso sta controllando se la parola è quella corretta (confrontando i due "tag"). Il processore riceve quindi una parola nello stesso momento (se non addirittura in anticipo) in cui viene a sapere se la parola è effettivamente quella richiesta.

Questo schema di corrispondenza inserisce linee di memoria consecutive in elementi della cache consecutivi. Nella cache è possibile memorizzare effettivamente $n \cdot k$ kbyte di dati contigui, dove n è il numero degli elementi che la cache può contenere (es. 2048) e k la dimensione delle linee (es. 32 byte); sia quindi $n \cdot k$ pari a 65.536 byte. All'interno di una cache non è possibile memorizzare allo stesso tempo due linee i cui indirizzi differiscano di esattamente 64 kbyte o di un multiplo di questo valore, dato che il loro campo `LINE` è uguale. Ad esempio, se un programma accede a dati che si trovano sulla locazione X e successivamente esegue un'istruzione che richiede dati presenti alla locazione X + 65.536 (o in qualsiasi altra locazione della stessa linea), la seconda istruzione ci obbligherà a ricaricare l'elemento della

cache, sovrascrivendo quello precedente. Tuttavia è da notare che questo tipo di collisioni avvengono raramente [2].

1.2.2 Cache set-associative

In questo modello si consente che ciascun elemento della cache disponga di due o più linee. Una cache con n possibili elementi per ciascun indirizzo è chiamata **cache associativa a n vie**.

Quando si effettua una richiesta alla cache, è necessario controllare un insieme di n elementi per vedere se la linea richiesta è presente nella cache. Questa verifica deve essere effettuata molto rapidamente; dati empirici hanno mostrato che cache a 2 o 4 vie garantiscono prestazioni sufficientemente buone da giustificare l'aumento dei componenti digitali richiesti [2].

La cache non è più divisa in 2^n linee di cache, come avveniva prima, ma in 2^n *cache set*; il numero di insiemi è limitato al numero di vie. Ogni set della cache ha m modi per fornire posizioni di salvataggio per m indirizzi congruenti. In figura 1.2 possiamo vedere una cache a 2 vie.

Nelle cache set-associative bisogna decidere algoritmicamente quale linea scartare quando se ne introduce una nuova; un algoritmo può essere ad esempio **LRU** (Least Recently Used).

1.3 Predizione dei salti

I calcolatori moderni sono organizzati a pipeline, ovvero sono capaci di dividere l'esecuzione di un'istruzione in più fasi (fetching dell'istruzione, decodifica dell'istruzione, fetching degli operandi, etc.) e sono in grado di eseguire queste diverse fasi nello stesso ciclo di clock; in altre parole, se al ciclo di clock 1 ho eseguito il fetch dell'istruzione 1, al ciclo di clock 2 posso contemporaneamente eseguire il fetch degli operandi dell'istruzione 1 e il fetch dell'istruzione 2, senza dover aver prima eseguito completamente tutti gli stadi per l'istruzione 1.

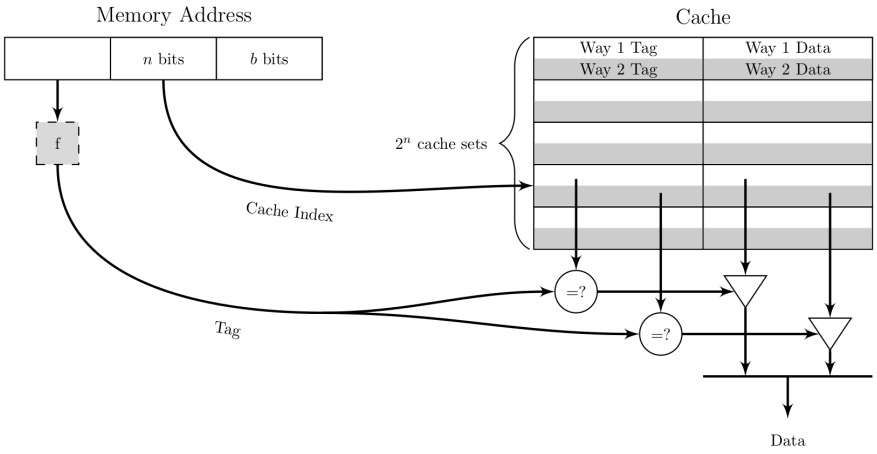

Figura 1.2: Cache set-associativa a 2 vie. I n bit intermedi sono l'indice della cache e servono a selezionare il set della cache. Il tag è usato per controllare tutte le vie simultaneamente. I dati nella via corrispondente sono restituiti al core in esecuzione.

L'uso della pipeline funziona in modo ottimale su codice lineare; ma questo modello purtroppo non è realistico, a causa delle istruzioni di salto, come quella nel seguente codice:

```
if (i == 0)            CMP i,0
    k == 1             BNE Else #salta a Else se i diverso da 0
else                   MOV k,1 #sposta in k il valore 1
    k == 2             BR Next # salta a Next
                       MOV k,2
```

Nell'esempio abbiamo due tipi di salto, incondizionato (`BR Next`) e condizionato (`BNE Else`). Per entrambi l'organizzazione a pipeline presenta problemi: infatti, siccome la decodifica dell'istruzione si può svolgere in un momento diverso dal fetching, l'unità di prelievo dovrebbe decidere dove prelevare l'istruzione successiva prima ancora di conoscerne il tipo. La conseguenza è che un buon numero di macchine esegue l'istruzione che *segue* un salto incondizionato, anche se logicamente non dovrebbe farlo. Un compilatore

ottimizzato inserisce istruzioni utili nella **posizione di ritardo**, ovvero la posizione immediatamente successiva un salto incondizionato.

Nel caso dei salti incondizionati, invece, la macchina non sa nemmeno se deve effettuare il salto. Siccome dovrebbe rimanere in stallo (e in un programma dove il 20% delle istruzioni sono salti condizionati significherebbe rimanere in stallo per tre o quattro cicli ad ogni salto condizionale), l'alternativa è provare a **predire** se la diramazione verrà effettuata o meno, ogni qualvolta se ne incontri una.

Ci sono due modelli principali di predizione dei salti.

1.3.1 Predizione statica delle diramazioni

Fatta in parte dall'hardware e in parte dal compilatore, predice sempre lo stesso esito per lo stesso salto durante tutta l'esecuzione del programma. Semplici meccanismi di predizione costante dell'hardware possono essere:

- predici sempre di non prendere la diramazione
- predici sempre di prendere la diramazione
- predici sempre di prendere i salti all'indietro e mai i salti in avanti

Per quanto riguarda l'ultimo punto, la direzione del salto, ovvero avanti o indietro, dipende dal valore dell'indirizzo della prossima istruzione da inserire nel program counter (la destinazione del salto), che può essere maggiore o minore rispetto all'indirizzo dell'istruzione attuale; nel primo caso si ha un salto in avanti, nel secondo all'indietro.

Un bit nel codice operativo del salto permette al compilatore di decidere la direzione della predizione. [3, 4]

1.3.2 Predizione dinamica dei salti

Nella **predizione dinamica dei salti** l'hardware influenza la predizione mentre l'esecuzione del programma procede. La predizione è decisa sulla base

della storia delle computazioni del programma: durante la fase di avvio, dove una predizione statica delle diramazioni può rivelarsi più efficace, si iniziano a raccogliere informazioni sulla storia dell'esecuzione del programma e, mano a mano che si raccolgono dati, le predizioni dinamiche diventano più accurate.

Per implementare la predizione dinamica quindi occorrono dei meccanismi hardware; questi ultimi sfruttano il comportamento a run-time dei salti per fare predizioni più accurate rispetto alle predizioni statiche. La storia delle computazioni del programma che viene considerata consiste proprio nei risultati delle occorrenze dei precedenti salti (cioè se un branch già incontrato è stato preso o meno, e in che direzione). Queste informazioni sono usate per provare a predire, dinamicamente, il risultato del salto corrente.

La componente che si occupa delle predizioni è la **BPU**, Branch Prediction Unit. La BPU è composta da due elementi principali: una **BHT**, Branch History Table (a volte viene usata una **PHT**, Pattern History Table) e una **BTB**, Branch Target Buffer.

La BHT (chiamata anche **BHB**, Branch History Buffer) è una tabella indicizzata sui bit meno significativi dell'indirizzo delle istruzioni di salto recenti, a cui sono associati uno o più bit per indicare se la diramazione dovrà essere presa o meno.

La PHT invece è una tabella bidimensionale, che contiene sempre dei contatori per determinare se effettuare o meno il salto, ma sarà indicizzata su due valori: oltre a usare i k bit meno significativi dell'indirizzo del branch si possono usare informazioni provenienti da altre tabelle, come la **Global History Table** o la **Local History Table**, che contengono informazioni rispettivamente sull'ultimo salto effettuato (per qualsiasi salto) e sugli ultimi salti effettuati per un branch specifico [3]. Queste informazioni consistono nel *pattern* dei salti, una stringa di bit di lunghezza n che indica appunto se gli ultimi n salti (per un branch specifico o in generale, a seconda che sia nella GHT o nella LHT) sono stati seguiti. Si dirà di più sui pattern a breve.

Il **BTB** infine associa, agli indirizzi di salto in esso salvati, gli indirizzi

di destinazione da inserire sul program counter. Questa tabella inoltre ha anche un importante vantaggio: se durante la fase di fetch la CPU "scopre" che l'indirizzo appena prelevato è salvato nella BTB, prima ancora della fase di decode saprà che si tratta di un'istruzione di salto.

Sono stati proposti diversi schemi di predizione dinamica, di seguito illustrati.

1.3.3 One level (o bimodale) e two level

Lo schema one level è il più semplice: fa un uso diretto della BHT, dove per ogni indirizzo è associato un contatore. Tipicamente si usano dei *contatori a saturazione* a due bit: in altre parole, questi contatori rappresentano un automa a stati finiti e per ognuna delle 4 combinazioni che si ottengono dai due bit corrisponde uno stato, il quale infine indica se effettuare la diramazione o meno, come illustrato in figura 1.3.

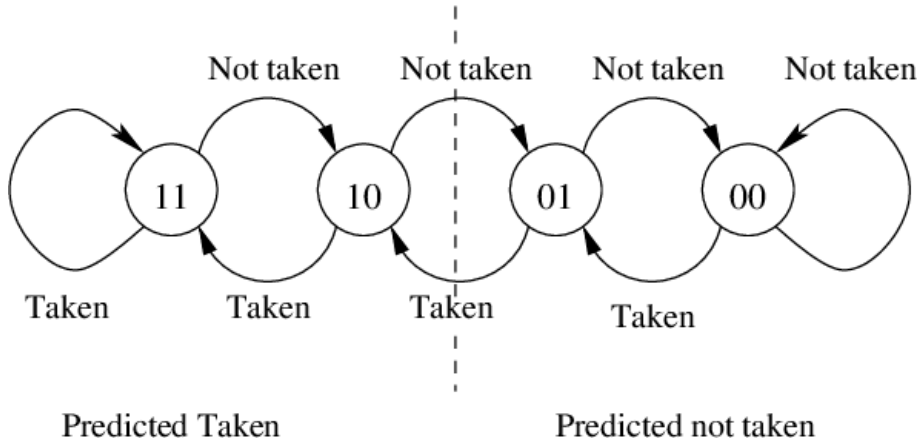

Figura 1.3: Automa a stati finiti per contatori a saturazione. Se il branch viene preso, incrementa il contatore; altrimenti lo decrementa. Se quando si incontra il branch siamo in uno dei due stati "predicted taken", allora eseguiamo il salto; altrimenti prediciamo che il salto non è da fare.

Nei predittori di livello 2 si tiene conto del fatto che un salto in un branch avviene anche a seconda del comportamento di altri branch. Si userà quindi

una PHT. In generale, un predittore di livello 2 è più preciso rispetto ad un predittore di livello 1, ma questo si ottiene con un'implementazione più costosa e una **fase di riscaldamento** più lunga, ovvero il tempo che deve passare prima di ottenere dei valori utili dal predittore.

Il funzionamento è il seguente: per ogni branch, c'è un registro di correlazione associato, ovvero un registro che unisce l'indirizzo del branch in esame con l'esito dell'ultimo branch. Quest'unione è detta *pattern*. Ad ogni pattern, è associata una voce nella **GPT**, Global Pattern History Table: ogni stesso pattern degli esiti dell'ultimo branch (eg. 1010 - il branch è stato preso nell'ultima esecuzione, non è stato preso nella penultima etc.) è associato nella stessa voce della tabella, indipendentemente dall'indirizzo attuale del branch. Per effettuare una predizione, si usano i k bit meno significativi dell'indirizzo di un branch per accedere al pattern nei registri di correlazione, e usando il pattern si cercano i bit di predizione nella GPT.

1.3.4 two level prediction di McFarling con condivisione dell'indice (gshare)

Nel 1993 McFarling suggerì questo schema di livello 2, che consiste nell'usare l'indirizzo dell'istruzione del branch e la storia del branch (realizzata con uno shift register - ovvero il pattern introdotto nella sezione 1.3.3). Queste due stringhe di bit sono unite facendone lo XOR; il risultato è usato per indicizzare la tabella dei bit di predizione. La figura 1.4 fornisce un'illustrazione del funzionamento di questo predittore.

1.3.5 Predittori ibridi o a "tournament"

Questo tipo di predittori, proposti sempre da McFarling, sono costruiti usando due predittori che lavorano indipendentemente l'uno dall'altro, e un selettore che sceglie quale delle predizioni dei due predittori usare. Una tabella poi memorizzerà i successi dei predittori. Se entrambi i predittori deducono la stessa scelta, non c'è ambiguità; ma se le predizioni sono discor-

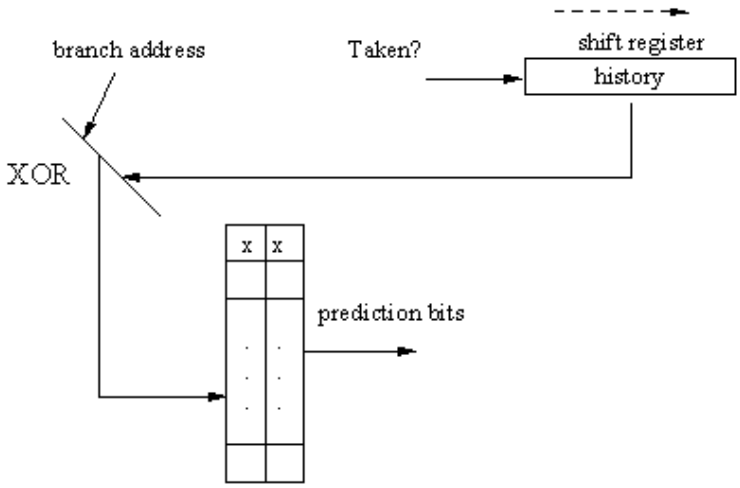

Figura 1.4: Predittore gshare.

danti, in tal caso si consulta la tabella (che può essere implementata usando i contatori a saturazione già visti), e aggiornandola se la previsione si rivela corretta o meno.

L'idea di usare due predittori deriva dal fatto che i predittori di livello 1 si comportano meglio all'inizio dell'esecuzione di un programma, quando la storia dei branch deve ancora essere inizializzata; ma i predittori di secondo livello riescono ad ottenere performance molto migliori con alcuni tipi di programmi, e.g. programmi con molti cicli. Solo usandoli entrambi si riescono a combinare i benefici dei due schemi.

1.4 Memoria Virtuale

La memoria virtuale è la tecnica che permette l'esecuzione di processi che non sono completamente in memoria. Permette di eseguire in concorrenza

processi che nel loro complesso (o anche singolarmente) hanno necessità di memoria maggiore di quella disponibile. Questo è possibile perché, mentre le istruzioni da eseguire e i dati su cui tali istruzioni operano devono essere in memoria, non è necessario che l'intero spazio di indirizzamento logico di un processo sia in memoria. I processi infatti non utilizzano tutto il loro spazio di indirizzamento contemporaneamente (es. routine di gestione di errori o di eccezioni particolarmente rare non hanno bisogno di rimanere nella memoria centrale fino a che non vengono chiamate - cosa che potrebbe anche non succedere mai).

Nei moderni sistemi operativi quando un processo viene lanciato occorre che avvenga il *binding* tra le istruzioni e gli indirizzi di memoria in cui queste saranno salvate. Ogni processo inoltre ha uno **spazio di indirizzamento logico**, ovvero un insieme di indirizzi a cui il processo fa riferimento; e uno **spazio di indirizzamento fisico**, un insieme di indirizzi fisici a cui ad ognuno corrisponde un indirizzo logico. Una componente fondamentale che si occupa di tradurre gli indirizzi logici in indirizzi fisici è la **MMU** (**M**emory **M**anagement **U**nit). La MMU svolge un'altra importante funzione: ovvero è in grado di implementare dei meccanismi di protezione della memoria, controllando che un processo non cerchi di accedere ad un indirizzo di un altro processo (i.e. ad un indirizzo appartenente allo spazio di indirizzamento logico di un altro processo) e fornendo quindi isolamento tra i processi.

Tornando alla memoria virtuale, ogni processo ha uno **spazio di indirizzamento virtuale** che può essere *più grande* di quello fisico. Gli indirizzi virtuali possono essere mappati su indirizzi fisici della memoria principale, oppure possono essere mappati anche sulla memoria secondaria. Nel caso si tenti di accedere ad un indirizzo virtuale mappato sulla memoria secondaria, i dati associati vengono trasferiti in memoria principale e, se la memoria è piena, si sposta in memoria secondaria i dati contenuti nella memoria principale che sono considerati meno utili.

Ci sono due tecniche fondamentali per fare memoria virtuale. La prima è quella della **paginazione**: lo spazio degli indirizzi virtuali è diviso in un

certo numero di **pagine** della stessa dimensione, dove una pagina è una parte di programma. La dimensione di una pagina è sempre una potenza di 2, per esempio 2^k, in modo tale che tutti gli indirizzi possano essere rappresentati con k bit. Anche lo spazio di indirizzamento fisico è suddiviso in porzioni che hanno la stessa dimensione di una pagina, di modo che ogni pezzo della memoria principale possa contenere esattamente una pagina. Un'area di memoria della grandezza di una pagina è chiamata *frame*. Ogni pagina ha un numero; una struttura dati chiamata *tabella delle pagine* mantiene la corrispondenza tra numero della pagina e indirizzo virtuale. Come descritto sopra, la MMU è poi in grado di convertire questi indirizzi virtuali in indirizzi fisici.

L'altra tecnica è la **segmentazione**, che permette di dividere il programma non più in parti uguali, ma in **segmenti**, ognuno dei quali è indipendente da un altro e consiste in uno spazio di indirizzamento. In questo modo si può avere uno spazio per la tabella dei simboli di un programma (contenente i nomi e gli attributi delle variabili), uno per il codice sorgente, uno per l'albero sintattico del programma, etc. Un segmento è un'entità logica, visibile al programmatore. Il segmento viene tipicamente diviso in pagine, che vengono poi gestite come precedentemente spiegato. I segmenti offrono poi una importante caratteristica: infatti, è possibile definire modalità di protezione diverse su segmenti diversi. Il sistema operativo può specificare privilegi di accesso, un offset e una lunghezza per ciascun segmento. Processi diversi usano segmenti diversi, e quindi lavorano su parti diverse di memoria fisica.

L'implementazione della memoria virtuale si appoggia in larga parte su componenti hardware (come ad esempio la sopracitata MMU). La traduzione degli indirizzi è estremamente critica per le performance, e il processore deve essere in grado di gestire la struttura dati che mantiene le corrispondenze tra indirizzi; conviene quindi usare una struttura simile ad un semplice array. Tuttavia, consideriamo una memoria virtuale di dimensione 2^{48} bit, con ogni

pagina di $8 * 2^{12} = 4$ KB; la memoria virtuale dovrà contenere

$$\frac{2^{48}}{2^{12}} = 2^{36} = 64 * 10^{9}\text{pagine}$$

La tabella inoltre dovrà avere una voce per ognuna di queste pagine, supponiamo di 64 bit. In conclusione, la dimensione della nostra struttura dati dovrà essere di $64 * 10^9 \times 8$ Byte = 64 GB. Ovviamente, questa non è una soluzione pratica. Si usano allora **tabelle di traduzione multilivello**, per mappare lo spazio di indirizzi virtuale in modo meno compatto nella memoria fisica. Usando più livelli di traduzione, e quindi più tabelle, la dimensione delle associazioni si può ridurre ad un overhead trascurabile.

Per tradurre un indirizzo virtuale in un indirizzo fisico il processore deve anzitutto cercare la tabella di traduzione di livello più alto, il cui indirizzo è memorizzato in un registro del processore.

I moderni processori Intel hanno 4 livelli di tabelle di traduzione, come mostrato in figura. Il livello più alto è il 4, detto PML4 (page map level 4), che divide lo spazio di indirizzamento virtuale di 48 bit in 512 regioni da 512 GB (*PML4 entries*). Le voci PML4 corrispondono a tabelle di puntatori di directory delle pagine (PDPT, page directory pointer table), delle tabelle che consistono in un insieme di indirizzi che puntano a loro volta a directory delle pagine. Quindi non si può mappare una pagina da 512 GB.

Ogni voce PML4 definisce delle proprietà della corrispondente regione di 512 GB, e ogni PDPT ha 512 voci, che a loro volta definiscono delle proprietà di regioni di memoria virtuale di 1 GB. Questa pagina virtuale da 1 GB può essere associata ad una pagina da 1 GB, oppure ad una directory delle pagine (PD, page directory). Ogni processo ha una directory delle pagine associata; le PD hanno 512 voci associate, che possono puntare a pagine da 2MB (usate tipicamente per mappare file, o grandi array), oppure ad una tabella delle pagine (PT, Page Table). Ogni PT ha 512 voci, ognuna delle quali punta ad una pagina di 4 KB. La situazione è riassunta nella figura 1.5.

Un altro elemento hardware che viene utilizzato nell'implementazione è la **TLB**, Translation Lookaside Buffer, una tabella che agisce come una ca-

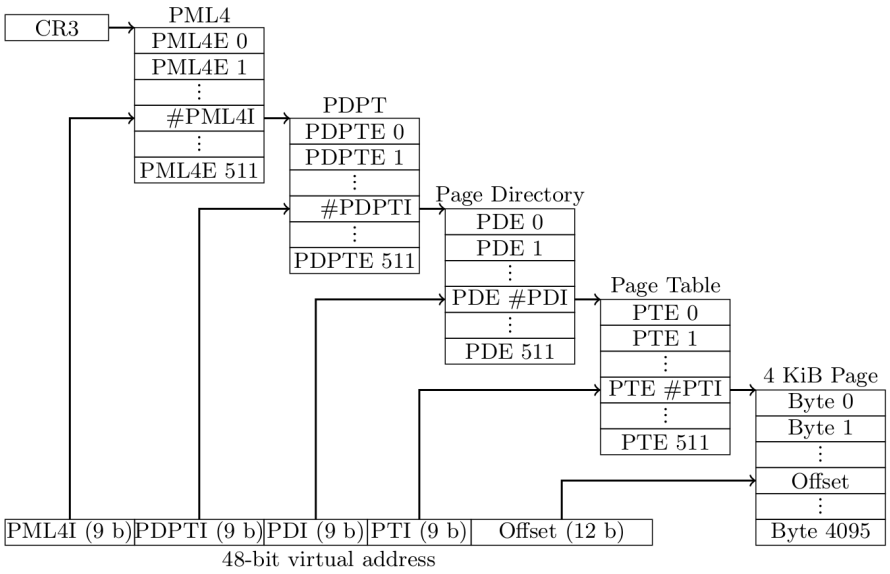

Figura 1.5: Traduzione delle pagine di 4 KB su un processore x86-64. `CR3` è il registro che mantiene l'indirizzo della PML4.

che e mantiene in una memoria accessibile velocemente dalla MMU alcune corrispondenze tra indirizzi virtuali e indirizzi fisici.

1.5 Esecuzione fuori sequenza

La maggior parte delle CPU moderne funziona sia a pipeline sia in modo superscalare (ovvero è in grado di eseguire più istruzioni per ciclo di clock). Ciò comporta in generale la presenza di un'unità di fetch che preleva istruzioni dalla memoria prima che siano necessarie, in modo da alimentare un'unità di decodifica. L'architettura più semplice prevede che tutte le istruzioni siano eseguite nell'ordine in cui siano prelevate dalla memoria (ipotizzando momentaneamente che l'algoritmo di predizione delle diramazioni non sbagli mai), tuttavia l'esecuzione in ordine non sempre fornisce prestazioni ottimali a causa delle dipendenze tra istruzioni: le istruzioni che richiedono un valore calcolato da un'istruzione precedente non possono essere eseguite fintanto che

quell'istruzione non abbia prodotto il valore desiderato (dipendenze RAW - Read After Write). In tali situazioni le istruzioni devono aspettare.

Nel tentativo di aggirare questi problemi e ottenere prestazioni migliori, alcune CPU permettono di saltare le istruzioni dipendenti per raggiungere le istruzioni successive che non lo sono. Naturalmente, l'algoritmo interno di scheduling delle istruzioni impiegato deve garantire che l'esecuzione produca lo stesso risultato che produrrebbe l'esecuzione ordinata.

1.5.1 Rinomina dei registri e ritiro delle istruzioni

Quando le istruzioni sono prelevate dal ciclo FDE, queste possono venire scomposte in istruzioni più semplici, dette micro-operazioni o μ-ops.

Un'istruzione che viene completata, i.e. eseguita dalla CPU, deve scrivere il valore del risultato della computazione in un registro; quando un'istruzione viene completata si può *ritirare*; si può quindi fare un "commit" e salvare le modifiche nella memoria centrale [23].

È fondamentale che, anche se le istruzioni sono lanciate fuori sequenza, siano ritirate in ordine, ovvero che non sia possibile ritirare l'istruzione k+1 se non è stata ritirata l'istruzione k; questo perché in caso venisse sollevato un interrupt, sarebbe difficile salvare lo stato della macchina per ripristinarlo successivamente. In particolare, non sarebbe possibile se siano state eseguite tutte le istruzioni fino ad un certo indirizzo e non quelle successive.

Come accennato sopra, l'esecuzione fuori sequenza può introdurre alcune dipendenze:

1. **Dipendenza WAR** (Write After Read): un'istruzione cerca di sovrascrivere un registro che un'istruzione precedente potrebbe non aver terminato di leggere;

2. **Dipendenza WAW** (Write After Write): alcune istruzioni utilizzano simultaneamente gli stessi registri, con il rischio che un'istruzione cancelli il risultato di un'altra istruzione.

Per eliminare queste dipendenze si può usare una tecnica detta **rinomina dei registri**. L'idea è che il processore utilizzi dei registri *nascosti*, in modo tale che se più istruzioni vogliono usare lo stesso registro, possono essere lanciate in modo concorrente, e una di esse potrà scrivere il risultato in un registro segreto S1 al posto di R1. R1 è rinominato come S1.

1.6 Esecuzione speculativa

I programmi sono divisi in **blocchi elementari**, ciascuno dei quali è costituito da una sequenza lineare di istruzioni con un punto di ingresso all'inizio e un punto di uscita alla fine. Un blocco elementare non contiene al suo interno alcuna struttura di controllo (come i costrutti `if` o `while`) di modo che la sua traduzione in linguaggio macchina non presenti nessuna diramazione. Il collegamento tra diversi blocchi elementari è costituito dalle strutture di controllo. Un programma in questa forma può essere rappresentato attraverso un grafo orientato.

La maggior parte dei blocchi elementari è di breve lunghezza e quindi non vi è al loro interno sufficiente parallelismo da poter sfruttare in modo efficacie. L'idea quindi per migliorare questa situazione è permettere al riordinamento di superare i limiti che separano i blocchi elementari: il guadagno maggiore si ottiene quando un'istruzione potenzialmente lenta può essere spostata più in alto nel grafo in modo che la sua esecuzione possa iniziare in modo anticipato. Questa potrebbe essere un'istruzione LOAD, un'operazione in virgola mobile, oppure l'inizio di una lunga catena di dipendenze. La tecnica di anticipare l'esecuzione del codice prima di una diramazione è nota come slittamento.

L'esecuzione di parti del codice prima ancora di sapere se saranno effettivamente richieste è chiamata **esecuzione speculativa**. Per poter usare questa tecnica sono necessari sia il supporto da parte del compilatore sia alcune estensioni dell'architettura. È compito del compilatore spostare esplicitamente le istruzioni, dato che un loro riordinamento che scavalchi i limiti dei blocchi elementari è generalmente al di là delle possibilità dell'hardware.

Supponiamo che un'istruzione, come la `LOAD`, causi un'eccezione. In questo caso, se una parola non è presente nella cache (e quindi si verifica appunto un'eccezione) mettere la macchina in attesa per una parola che potrebbe non essere necessaria è controproducente; alcune macchine moderne, per ovviare a questo problema, utilizzano un'istruzione speciale, `SPECULATIVE-LOAD`, che cerchi di prelevare la parola dalla cache, ma che non compia alcuna azione se non la trova.

È essenziale che nessuna delle istruzioni speculative produca risultati irrevocabili, dato che in un secondo momento si potrebbe scoprire che non dovevano essere eseguite. Un modo comune per evitare che una parte del codice speculativo scriva nei registri prima ancora di sapere se ciò è voluto, consiste nel rinominare tutti i registri di destinazione utilizzato dal codice speculativo. In questo modo vengono modificati solo i registri di lavoro, così che non sorgano problemi se in ultima istanza il codice risulta non richiesto. Se al contrario il codice è necessario, allora i registri di lavoro vengono copiati nei registri di destinazione corretti.

Immaginiamo la seguente situazione:

```
if (x > 0) z = y/x;
```

dove x, y e z sono variabili in virgola mobile. Si supponga che tutte le variabili siano prelevate e memorizzate in anticipo nei registri e che venga usata la tecnica dello slittamento del codice per spostare l'esecuzione della lunga operazione in virgola mobile prima del test if. Sfortunatamente x vale 0 e l'eccezione generata dalla divisone per 0 fa terminare il programma. Il risultato è che l'uso dell'esecuzione speculativa ha fatto fallire un programma corretto. La cosa peggiore è che ciò è avvenuto nonostante il programmatore avesse previsto il problema e programmato esplicitamente il codice in modo da evitarlo. Una possibile soluzione consiste nell'avere delle versioni speciali delle istruzioni che possano sollevare delle eccezioni, e nell'aggiungere a ogni registro un bit chiamato **poison bit** ("bit avvelenato"). Quando un'istruzione speculativa fallisce, essa, al posto di causare un'eccezione, assegna il

valore 1 al poison bit del registro risultato. Se in un secondo momento un'istruzione regolare utilizza questo registro allora viene sollevata un'eccezione (com'è giusto che avvenga). In ogni caso, se il risultato non è mai utilizzato, allora il poison bit può essere cancellato senza provocare alcun danno.

1.7 DRAM

La RAM dinamica (DRAM) è un tipo di memoria che può essere letta e scritta in modo non sequenziale; è costruita con un condensatore e un transistor per bit. Il condensatore può essere caricato o scaricato per memorizzare i valori 0 oppure 1. La carica elettrica dura però pochi millisecondi, di conseguenza è necessario effettuare periodicamente un'operazione di refresh. Le DRAM hanno lo svantaggio di essere più lente (decine di nanosecondi) rispetto alla loro controparte statica, e hanno inoltre un'interfaccia più complessa; ma hanno il vantaggio di avere una densità di bit per chip maggiore, e offrono quindi maggiore capacità ad un costo più contenuto.

La memoria principale dei moderni computer è tipicamente DRAM. La DRAM ha una latenza significativamente più alta rispetto alle varie cache presenti all'interno del processore. La ragione della latenza risiede non solo nella più bassa frequenza di clock delle celle DRAM, ma anche in come la DRAM è organizzata e come è collegata al processore. Siccome è difficile ridurre la latenza, i produttori preferiscono concentrarsi nell'accrescere la bandwitdh della DRAM. La maggior bandwitdh può essere utilizzata per nascondere la latenza, ad esempio attraverso il prefetching speculativo. I processori moderni hanno un controller della memoria sul chip che comunica attraverso il bus della memoria della DRAM.

La figura 1.6 illustra un sistema molto semplice con un singolo vettore di DRAM, collegato al processore attraverso un bus di memoria. Questo array di DRAM consiste di *righe* e *colonne* di DRAM (tipicamente 2^{10}). I sistemi moderni organizzano la memoria in modo che una riga di DRAM abbia una

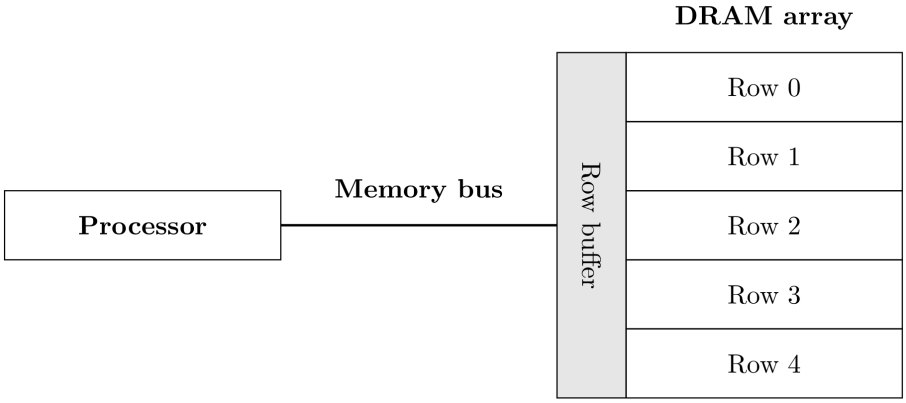

Figura 1.6: Un sistema molto semplice, con un singolo array DRAM, connesso al processore tramite un bus della memoria.

dimensione di 8KB. Una riga può essere "aperta" o "chiusa": se è aperta, l'intera riga è mantenuta nel *row buffer*.

Per ottenere un valore dalla DRAM, il processore invia una richiesta al controller della memoria integrato. Il controller della memoria determina poi una sequenza di comandi da spedire sul bus della memoria alla DRAM per reperire i dati. Se la riga attualmente aperta contiene i dati richiesti, il controller della memoria li reperisce semplicemente dal row buffer. Questa situazione è chiamata *row hit*. Se la riga aperta non contiene i dati richiesti, siamo nella situazione così detta "row conflict". Il controller della memoria per prima cosa chiude la riga corrente, cioè riscrive l'intera riga sulle celle effettive della DRAM. Successivamente attiva la riga con i dati da reperire, che è poi caricata nel row buffer. Poi il controller della memoria recupera i dati dal row buffer.

Possiamo vedere che il row buffer della DRAM in figura 1.6 si comporta in modo identico alla cache a indirizzamento diretto. Così come (in tale modello di cache) accessi congruenti alla memoria portano costantemente a cache miss, accessi alterni alle righe portano costantemente a row conflicts nella DRAM. Per aumentare le performance delle DRAM ed eliminare i colli di bottiglia sono stati adottati approcci simili a quelli usati nel caso delle

cache. Le DRAM moderne sono organizzate in **canali**, **DIMM** (Dual Inline Memory Module), **rank** e **banchi**.

Nell'esempio citato (cfr. 1.6), abbiamo solamente 1 canale, 1 DIMM, 1 rank e 1 banco. Una moderna memoria DRAM DDR3 ha 8 banchi, e una DDR4 perfino 16 banchi (per rank). Ognuno di questi banchi ha uno stato indipendente e quindi può avere diverse righe aperte allo stesso tempo. Questo riduce significativamente le possibilità di row conflict per accessi non casuali alla memoria. I moderni moduli DRAM hanno tipicamente da 1 a 4 rank, moltiplicando ulteriormente il numero dei banchi. In modo simile, i sistemi moderni spesso permettono di installare moduli DIMM multipli, moltiplicando ancora il numero dei banchi con il numero dei DIMM. Infine, il controller della memoria riordina gli acessi alla memoria per ridurre il numero dei row conflict.

Il parallelismo spaziale riduce il numero il numero dei row conflicts. Quindi, non influenza direttamente la bandwidth, ma la latenza media. Per aumentare il parallelismo spaziale e di conseguenza direttamente la bandwidth, i sistemi moderni impiegano canali multipli. Ogni canale è gestito indipendentemente e in parallelo sul bus della DRAM. Questo moltiplica efficacemente la bandwidth per il numero dei canali.

Due indirizzi possono essere fisicamente adiacenti nello stesso chip DRAM solo se sono mappati nello stesso DIMM, rank e banco. In questo caso i due indirizzi sono nello stesso banco. Se due indirizzi sono mappati nello stesso numero di banco, ma in un diverso rank o DIMM, non sono nello stesso banco e quindi generalmente non fisicamente adiacenti nel chip DRAM.

Esistono funzioni che mappano dagli indirizzi fisici a: canali, DIMM, rank e banchi. Mentre AMD documenta publicamente queste funzioni di indirizzamento, Intel no. Tuttavia queste funzioni sono state recentemente scoperte grazie al reverse-engineering; la conoscenza delle funzioni di indirizzamento della DRAM permette attacchi side-channel basati sulla DRAM [6].

Capitolo 2

Attacchi microarchitetturali automatizzati

Gli attacchi microarchitetturali sono una tipologia di attacchi che sfrutta vulnerabilità in elementi microarchitetturali; in questa sezione si parlerà di attacchi automatizzati, ovvero che sono basati su software, senza avere accesso all'elaboratore fisico.

Questi attacchi possono basarsi su elementi diversi; i principali attualmente sono:

1. cache;
2. predittore dei salti;
3. esecuzione speculativa;
4. DRAM.

2.1 Attacchi basati sulla cache

Siccome l'idea alla base delle memorie cache è accedere alle informazioni più velocemente, ci possono essere delle differenze temporali osservabili tra il recuperare un dato dalla memoria centrale e un dato dalla memoria cache.

Sfruttando questa intuizione, nel 1996 Paul C. Kocher mostrò come fosse possibile trovare gli esponenti Diffie-Hellman e rompere la cifratura RSA (cfr. appendice B) sfruttando queste differenze temporali [5]; questi attacchi sono detti **cache timing attacks** [6].

I primi attacchi avevano come target algoritmi crittografici, e solo più tardi sono stati generalizzati in *Evict + Time*, *Prime + Probe* [7], *Flush+Reload* [8]. Altri attacchi simili sono *Flush+Flush* e *Prime + Abort*, qui non trattati.

2.1.1 Evict + Time

Come nell'articolo originale [7], spieghiamo questo attacco contro l'algoritmo di criptazione AES. AES prende in input un blocco da cifrare e una chiave simmetrica, dopodiché esegue delle operazioni prestabilite sui bit da cifrare in base al valore della chiave. Le operazioni, effettuate in 10 round, sono basate su delle tabelle di lookup. Per motivi di efficienza, queste tabelle sono precalcolate dal programmatore oppure durante l'inizializzazione del sistema; di conseguenza, vengono caricate nella cache. Detta T_ℓ la tabella l-esima usata durante criptazione, consideriamo il seguente predicato:

$Q_k(p, \ell, y) = 1 \iff$ la criptazione con AES del testo in chiaro p con la chiave k accede il blocco di memoria di indice y in T_ℓ almeno una volta (nei 10 round).

Siccome l'indice che verrà acceduto nella tabella di lookup T_ℓ dipende dalla chiave, calcolando questo predicato possiamo avere informazioni sulla chiave. Ma come è possibile calcolarlo? Nella tecnica **evict+time** si manipola lo stato della cache prima di ogni criptazione, e poi si osserva il tempo impiegato per eseguirla. In questo attacco però è richiesta la capacità di innescare una criptazione e di sapere quando inizi e quando termini. In un attacco chosen-plaintext[1], i passaggi sono i seguenti:

1. Innesca la criptazione di **p**.

[1] Detto m_i l'i-esimo testo in chiaro e c_i il testo m_i cifrato con un algoritmo di cifrazione, in un chosen plaintext attack l'attaccante conosce l'algoritmo di cifrazione/decifrazione, il testo cifrato e una o più coppie $\{m_i, c_i\}$ con m_i scelto dall'attaccante

2. *(evict)* Accedi alcuni indirizzi di memoria W, ad almeno B bytes di distanza, che sono congruenti con la tabella di lookup T_ℓ.

3. *(time)* Innesca una seconda criptazione di **p** e cronometrala. Il tempo misurato è detto punteggio di misurazione.

Lo step 1. assicura che tutti i blocchi di memoria che vengono acceduti durante la criptazione di **p** sono nella cache. Lo step 2., se la cache è W-associativa, assicura che i blocchi siano rimossi dalla cache. A questo punto, al punto 3. quando inneschiamo una nuova criptazione, se il nostro predicato $Q_k = 1$ allora il blocco T_ℓ andrà ricaricato nella cache dalla memoria centrale; altrimenti la criptazione sarà più veloce. La differenza di velocità sarà quella di un cache hit vs. un cache miss.

La debolezza di questo metodo di misurazione è che, siccome deve innescare una criptazione e misurarne il tempo, è molto sensibile alle variazioni nell'operazione, questo perché tipicamente quando si innesca una criptazione si esegue anche del codice addizionale, e questo può causare del rumore nelle misurazioni.

2.1.2 Prime + Probe

Questo metodo cerca di scoprire quali sono i blocchi di memoria letti dalla criptazione di AES a posteriori, esaminando cioè lo stato della cache dopo la criptazione. Anche in questo caso l'attacco è diviso in 3 step principali:

0. Crea un array A

1. *(prime)* Leggi i valori dell'array.

2. Innesca una criptazione di **p**.

3. *(probe)* Rileggi l'array A.

Lo step 1. riempie la cache con i dati dell'attaccante; nello step 2, i dati sono sovrascritti parzialmente perché devono essere caricate le tabelle per

AES. A questo punto, rileggendo i nostri dati nello step 3, possiamo vedere quali dati sono stati sostituiti dalle tabelle (in quanto saranno stati appunto sostituiti e dovranno essere ricaricati dalla memoria centrale). Tutto ciò suppone che si conosca l'indirizzo di T_0 , la prima tabella.

Questo attacco è molto meno soggetto alle variazioni temporali rispetto al precedente.

2.1.3 Flush + Reload

Questo attacco, descritto da [8], è il più potente dei tre: infatti non ha la granularità dei set delle cache, ma bensì delle linee, il che lo rende più preciso; inoltre attacca il Last Level Cache, che è il più lontano dai core del processore e quindi è condiviso da più core; anche questo fatto conferisce maggiore precisione ai dati che si possono raccogliere [8].

L'attacco può essere eseguito su un core diverso da quello dove la vittima sta eseguendo il suo programma.

La tecnica flush+reload è una variante di prime+probe che si basa sulla condivisione delle pagine nella memoria (tecnica utilizzata per ottimizzare il consumo della memoria, dove appunto si condividono pagine di memoria identiche tra i processi che le stanno usando) tra la vittima e l'attaccante. Con le pagine condivise, la spia (attaccante) può verificare che una specifica linea di memoria sia rimossa dall'intera gerarchia della cache. La spia sfrutta ciò per monitorare l'accesso alla linea di memoria. Anche questo attacco si divide in tre fasi:

1. La linea di cache monitorata è rimossa dalla (gerarchia della) cache.

2. La spia attende un po' di tempo per far accedere alla vittima la linea di memoria.

3. La spia ricarica la linea di memoria, misurando il tempo di caricamento. Se nella fase 2. la vittima aveva acceduto alla linea, allora sarà già

presente nella cache e il tempo per compiere l'operazione sarà breve; vale il duale nel caso la vittima non avesse acceduto la linea nella fase 2.

Questo attacco è soggetto ad alcune perturbazioni: anzitutto, ci potrebbero essere delle sovrapposizioni tra la fase di reload dell'attaccante e l'accesso ai dati della vittima. Inoltre, diverse ottimizzazioni del processore a causa di accessi speculativi alla memoria potrebbero causare falsi positivi.

Questo attacco si basa sul fatto di poter rimuovere specifiche linee di memoria dalla cache, abilità che è garantita dall'istruzione `clflush`, che rimuove linee di cache da tutta la gerarchia. Rimuovere la linea da tutti i core garantisce che il prossimo accesso inserirà i dati nel Last Level Cache.

Questo attacco è stato usato in [8] per estrarre componenti della chiave privata dall'implementazione di GnuPG di RSA.

2.2 Attacchi basati sul predittore dei salti

Nel 2007 Acıiçmez et al. [9] hanno dimostrato come fosse possibile sfruttare il predittore dei salti per risalire alla chiave privata del cifrario RSA. La BPU (branch prediction unit) dispone di una cache, la BTB (branch target buffer), dove vengono salvati gli indirizzi di destinazione dei rami condizionali precedentemente incontrati durante l'esecuzione. Questa cache ha una dimensione limitata, quindi (come già visto) può essere necessario dover eliminare alcune voci precedentemente salvate per far posto a nuovi indirizzi.

I ricercatori hanno usato un'implementazione di RSA nella quale alcuni salti condizionali dipendevano dal valore della chiave (cfr. appendice A); di conseguenza, facendo quanto già visto per gli attacchi alla cache con la BTB (ovvero eliminando alcuni valori, o sostituendone con altri scelti dall'attaccante) è possibile notare una differenza temporale tra l'esecuzione di una criptazione/decriptazione e l'altra (a causa dei cache miss e hit); sapendo in corrispondenza di quali valori si hanno i rallentamenti (grazie alla manipolazione della BTB) e sapendo che il valore dei bit della chiave è legato

alla velocità di esecuzione (nell'implementazione scelta), è possibile scoprire il valore della chiave.

Nel 2018 è stato scoperto BranchScope, un altro attacco basato sul predittore direzionale dei salti[10]. Questo attacco non ha come target la BTB, ma attacca la componente direzionale del predittore (prendere o non prendere un branch). Come nell'attacco illustrato precedentemente, se in un algoritmo di criptazione/decriptazione la direzione di un salto condizionale dipende dal valore dei bit della chiave, sapendo in che direzione avverrà il salto possiamo risalire al valore della chiave. Vediamo nel dettaglio come funziona.

Anzitutto, l'attacco presuppone l'esistenza di un programma *vittima* e un programma *spia*. Il primo contiene informazioni che il secondo tenta di ottenere. Sono inoltre richiesti all'attaccante le seguenti capacità:

- Co-residenza sullo stesso core fisico: i programmi *vittima* e *spia* devono essere eseguiti sullo stesso core fisico, siccome la BPU è condivisa a livello di core. Come mostrato in altri lavori, esistono tecniche che rendono possibile forzare questa co-residenza [11].

- Slowdown della vittima: al fine di eseguire attacchi BranchScope ad alta risoluzione, dove si possono individuare il comportamento di un'esecuzione individuale di un branch, il processo vittima deve essere rallentato. Rallentare il processo vittima è un problema ortogonale che può essere risolto in diversi modi [11].

- Innescare l'esecuzione del codice della vittima: l'attaccante può iniziare l'esecuzione del codice del processo vittima in modo da costringere la vittima a eseguire l'operazione target vulnerabile in qualsiasi momento. Si pensi ad esempio ad un server che invia dati crittografati; l'attaccante può innescare una risposta da questo server inviandogli semplicemente una richiesta.

L'attacco si divide in tre parti:

- *Stage 1: Prime.* Il processo dell'attaccante imposta una voce nella PHT (Pattern History Table) della BPU in uno stato specifico. Questa inizializzazione è ottenuta eseguendo un blocco randomizzato e attentamente selezionato di istruzioni di salto. Questo blocco è generato a priori una sola volta dall'attaccante.

- *Stage 2: Esecuzione del processo vittima.* L'attaccante inizia l'esecuzione di un branch nel processo della vittima che intende monitorare e attende fino a che lo stato della PHT è modificato dall'attività della vittima.

- *Stage 3: Probe.* L'attaccante esegue, cronometrandole, altre istruzioni di salto che mirano alla stessa voce PHT della vittima per osservare i risultati delle loro predizioni. L'attaccante correla gli esiti delle predizioni con lo stato della PHT per identificare la direzione del branch della vittima.

L'attaccante deve essere in grado di causare collisioni tra i suoi branch e quelli del processo vittima nella PHT. Queste collisioni, sapendo come funziona il predittore, permettono all'attaccante di svelare la direzione del branch della vittima.

Se l'indicizzazione nella PHT è basata semplicemente sull'indirizzo delle istruzioni, come nei predittori di livello 1, creare collisioni nella PHT tra rami di due processi è semplice, siccome gli indirizzi virtuali del codice della vittima non sono tipicamente un segreto. Se viene usato ASLR (Address Space Layout Randomization) per rendere casuali le locazioni del codice, l'attaccante può usare attacchi side channel contro ASLR per togliere la randomicità [11].

Se il predittore in uso è ibrido, l'attaccante deve forzare l'uso del predittore di livello 1.

2.2.1 Inizializzare la PHT, forzare l'uso del predittore di livello 1 e strategia di probing

La logica di selezione nell'hardware cerca di scegliere il predittore più preciso. Una sequenza casuale ma ripetitiva di esiti della stessa istruzione di salto non può essere predetta accuratamente dal predittore di livello 1, ma può essere imparata eventualmente (i. e. dopo alcune esecuzioni, ovvero dopo che la sua storia è stata inizializzata) dal predittore di tipo gshare. Tuttavia, siccome il predittore gshare impiega più tempo per imparare un pattern, per i nuovi branch le cui informazioni non sono salvate nella storia del predittore viene usato il predittore di livello 1.

Sapendo ciò, per forzare la BPU a usare tale predittore, si può ciclare attraverso un numero di branch collocati ad indirizzi che sono in collisione con il branch della vittima nel predittore del branch (se anche quello è di livello 1), in modo che in qualsiasi momento il branch attualmente in uso nel processo attaccante non esista nella BPU, costringendo l'unità ad usare il predittore di livello 1.

Il compito più difficile è costringere il codice della vittima a usare il predittore di livello 1, sicché non è sotto il controllo dell'attaccante. L'unico modo che ha l'attaccante è: 1) assicurarsi che i branch usati nell'attacco non siano stati recentemente incontrati, e quindi cominciando la predizione per quei branch con il predittore di livello 1, oppure 2) rendere inaccurato il predittore di livello 2, costringendo il selettore a scegliere quello di livello 1. L'attaccante deve assicurarsi che almeno una delle due proprietà sia valida.

Questo obiettivo si può raggiungere sviluppando un blocco di codice con una sequenza con molti branch che l'attaccante esegue per abbassare la precisione del predittore di livello 2, e rimpiazzare i branch della vittima nella BPU. Come risultato, il codice della vittima userà il predittore di livello 1, permettendo all'attaccante di ottenere le collisioni. Infine, con questo codice si può anche inizializzare (*prime*) le voci nella PHT in uno stato desiderato che permetta all'attaccante di identificare in modo affidabile gli esiti del branch tramite il funzionamento della predizione dell'automa a stati.

Supponiamo di eseguire una sola istruzione di salto, senza nessuna storia di esecuzione, all'interno di un singolo processo. Anzitutto consideriamo l'automa a stati del predittore composto da un contatore a 2 bit, che rappresenta i 4 stati: strongly taken (ST), weakly taken (WT), weakly not taken (WN) e strongly not taken (SN). Primo, eseguiamo l'istruzione di salto appena menzionata tre volte per inizializzare (*prime*) la voce corrispondente nella PHT, mettendola in una dei due stati strong (NT o ST). Secondo, eseguiamo lo stesso branch un'altra volta con entrambi risultati "taken" e "not taken" (in due prove separate). Questo stato è chiamato *target*. Infine, eseguiamo lo stesso branch altre due volte cercando di rilevare le predizioni errate (stato *probing*). Durante questo stage, registriamo anche l'accuratezza delle predizioni per ognuno dei due branch di probe. Il nostro obiettivo è quello di identificare la direzione del salto del branch della vittima; questo si può fare nello stage di target: ad esempio, consideriamo il caso dove il branch è eseguito 3 volte con esito "not taken" (stage di *prime*). L'aspettativa è che questa azione porterà l'automa nello stato WN. Il branch è poi eseguito due volte nella fase di probe con esito "taken". In questo caso, il primo branch eseguito nella fase di probe verrà mal predetto, mentre il secondo sarà predetto correttamente. Al contrario, se il branch nella fase target aveva esito "not taken", l'automa rimarrà nello stato SN. In quel caso, entrambi i branch nella fase di probe saranno predetti male.

La tabella sottostante mostra come è possibile determinare uno stato nella PHT eseguendo due probe individuali con la stessa istruzione di salto, con esiti "taken" e "not taken".

2.2.2 Note implementative

Il processo dell'attaccante si basa su *hardware performance counters*, un insieme di registri del processore special-purpose, per identificare eventi di previsioni corrette e scorrette. Se l'accesso a performance counters non è di-

[2]MH è stato osservato su processori Intel con architettura Haswell e Sandy Bridge, mentre MM su Skylake.

Prime	**Stato dopo prime**	**Target**	**Stato dopo target**	**Probe**	**Osservazione**
TTT	ST	T	ST	TT	HH
TTT	ST	T	ST	NN	MM
TTT	ST	N	WT	TT	HH
TTT	ST	N	WT	NN	MH^2
NNN	SN	T	WN	TT	MH
NNN	SN	T	WN	NN	HH
NNN	SN	N	SN	TT	MM
NNN	SN	N	SN	NN	HH

Tabella 2.1: Transizioni della macchina a stati per una singola voce nella PHT. La voce è impostata in uno degli stati "strong" nella fase di prime, poi nella fase target viene eseguito un salto e lo stato risultante è registrato usando *performance counters* nella fase di probe. M = misprediction (predizione sbagliata), H = hit (predizione corretta), T = taken, N = not taken.

sponibile, si possono fare misurazioni temporali usando il *time stamp counter*, un registro che conta il numero di cicli avvenuti dall'ultimo reset. Nel primo caso, la spia può estrarre una sequenza dei valori dei salti predetti in modo scorretto e decodificare questa sequenza per determinare la direzione del branch della vittima. Per esempio, se l'attaccante osserva una sequenza di 2 salti mal predetti, oppure uno predetto correttamente e l'altro no, allora il branch della vittima è identificato come "taken", altrimenti è "not taken".

Per usare i performance counter, l'attaccante dovrebbe disporre di almeno privilegi parzialmente elevati. Un approccio alternativo per identificare eventi relativi ai branch è quello di osservare i loro effetti sulla performance della CPU: un branch scorrettamente predetto implicherà il fetching di istruzioni dal percorso sbagliato, perdendo quindi numerosi cicli per ricomin-

ciare la pipeline. Di conseguenza, l'attaccante può tracciare il numero di cicli per determinare se un branch è stato predetto correttamente. Questo cronometraggio può essere realizzato con alcune istruzioni dedicate (nei processori intel): `rdtsc` o `rdtscp`. Queste istruzioni forniscono i processi utente con un accesso diretto all'hardware che si occupa di tenere traccia dei cicli, bypassando altri layer di sistemi software.

2.2.3 Mitigare BranchScope

Sono state presentate alcune soluzioni per mitigare questo attacco:

- **Mitigazioni tramite software:** Le mitigazioni software non forniscono protezione dai *covert channel*[3], siccome non rimuovono la fonte del leak dei dati nell'hardware.

  Una possibile tecnica è di eliminare salti condizionali dai programmi target: questa tecnica, nota come *if-conversion*, è un'ottimizzazione del compilatore che converte branch condizionali in codice sequenziale usando istruzioni condizionali come `cmov`, trasformando effettivamente dipendenze di controllo in dipendenze dei dati.

  Non si sa se sia possibile però convertire applicazioni reali in codice "branch-free".

- **Difese hardware:** diversi meccanismi di difesa sono attuabili via hardware. Anzitutto, è possibile prevenire le collisioni nella PHT (uno dei requisiti principali di BranchScope) rendendo la PHT "casuale": la funzione di indicizzazione della PHT potrebbe prendere in input dei dati unici per l'applicazione software a cui appartengono i branch in uso, o anche usare dei numeri generati casualmente dal processo, per prevenire appunto le collisioni.

[3]Cioè il canale di comunicazione "nascosto", che consiste appunto nel poter leggere la direzione del salto del branch di un processo a causa della vulnerabilità appena discussa. Questi canali permettono a processi malevoli di comunicare tra loro di nascosto.

Un'altra soluzione potrebbe essere quella di rimuovere le predizioni per branch sensibili, soluzione che potrebbe essere attuata permettendo al programmatore di indicare nel suo software quali sono i branch che contengono dati sensibili. Inoltre, la CPU potrebbe comunque usare tecniche di predizione statica per quei branch. Questo metodo però richiede una sostanziale perdita di performance, e come le tecniche software non fornisce protezione contro l'attacco covert channel.

Infine, si potrebbe partizionare in qualche modo la BPU, in modo che l'attaccante e la vittima non condividano le stesse strutture.

2.3 Attacchi basati sull'esecuzione speculativa

L'esecuzione speculativa è una tecnica di progettazione delle microarchitetture usata per migliorare la velocità del processore; tramite essa, la CPU prova ad indovinare quali saranno le direzioni future del flusso di esecuzione, ed esegue in anticipo istruzioni in quei percorsi [12]. Nella sezione precedente abbiamo discusso di come queste direzioni sono predette; l'esecuzione speculativa si concentra invece sull'aspetto appunto dell'esecuzione, ovvero di cosa fare dopo che si è ottenuta una predizione dalla BPU.

L'esecuzione speculativa pone problemi dal punto di vista della sicurezza in quanto può indurre la CPU ad eseguire istruzioni che non sarebbero dovute essere eseguite durante una corretta esecuzione del programma: nell'esempio sottostante, se `password_utente` è scorretta, l'esecuzione speculativa potrebbe portare a eseguire comunque la funzione `login_utente()`, anche se il programma sorgente è logicamente corretto e non lo prevede.

```
if (password_utente == password_corretta){
        login_utente();
}
```

Le istruzioni che si rivelano scorrette e quindi vanno "annullate" riportando lo stato della computazione a prima della loro esecuzione, sono dette **istruzioni transitorie** [12]. Per mostrare come queste istruzioni transitorie possano essere sfruttate in un attacco side channel, introduciamo gli (la classe di) attacchi **Spectre**.

2.3.1 Spectre

Spectre è una classe di attacchi scoperta nel 2018 che, influenzando l'esecuzione delle istruzioni transitorie, sono in grado di rubare informazioni dallo spazio di indirizzamento della memoria della vittima. L'attacco è in grado di fare ciò sia da codice nativo (nell'esempio sottostante), sia da altre fonti, come ambienti SandBox (e.g. JavaScript) e perfino attraverso l'interprete eBPF.

- **Codice nativo** Consideriamo un semplice programma che contiene informazioni segrete nel suo spazio di indirizzamento di memoria. Cerchiamo il binario compilato e le librerie condivise del sistema operativo per sequenze di istruzioni che possono essere usate per rubare informazioni dallo spazio di indirizzamento della vittima. Possiamo scrivere un programma attaccante che sfrutti l'esecuzione speculativa per eseguire la sequenza precedentemente trovata come istruzioni transitorie: usando questa tecnica, siamo in grado di leggere memoria dallo spazio di indirizzamento della vittima, incluse le informazioni segrete ivi salvate.

Per montare un attacco Spectre, un attaccante inizia localizzando o introducendo una sequenza di istruzioni nello spazio di indirizzamento del processo che, quando eseguite, agiscono come un trasmettitore a canale nascosto (covert channel) che fa trapelare la memoria della vittima o ne registra i contenuti. L'attaccante poi inganna la CPU per farle eseguire speculativamente (ed erroneamente) questa sequenza di istruzioni. Infine, l'attaccante recupera le informazioni attraverso il covert channel. Mentre i cambiamenti allo stato nominale risultanti dall'esecuzione speculativa sono annullati,

informazioni precedentemente "fuoriuscite" o cambiamenti di stato di altri elementi microarchitetturali (come ad esempio il contenuto della cache) possono sopravvivere al processo di annullamento.

La descrizione appena fornita è quella dello schema generale dell'attacco, e deve essere concretizzata fornendo un modo per indurre l'esecuzione speculativa erronea, ed un canale nascosto microarchitetturale. I canali scelti nell'articolo originale si basano sulla cache (Flush + Reload, Evict + Reload) [12].

Varianti per influenzare l'esecuzione speculativa

Ci sono principalmente due tecniche per indurre l'esecuzione speculativa di codice malevolo nella CPU: sfruttare i salti condizionali oppure i salti indiretti. Sarà illustrata solo la prima variante.

Salti condizionali Consideriamo il caso in cui il codice nel listato 2.1 faccia parte di una funzione (e.g. una system call o in una libreria) che riceve un intero `x` non segnato da una fonte non sicura. Il processo che esegue questo codice ha accesso ad un array di byte non segnati `array1` di dimensione `array1_size`, e un secondo array di byte `array2` di dimensione 1MB.

```
if (x < array1_size)
    y = array2[array1[x] *  4096]
```

Listing 2.1: Esempio di salto condizionale

Supponiamo che l'attaccante riesca a eseguire 2.1 in modo tale che:

- il valore di x è scelto malignamente (*out-of-bound*), in modo tale che `array1[x]` punta ad un byte k segreto da qualche parte nella memoria della vittima;

- `array1_size` e `array2` non sono presenti nella cache, mentre k sì;

- la BPU è stata "male addestrata" (il codice è stato eseguito più volte con valori validi di x), in modo tale che l'`if` sarà vero in una prossima esecuzione.

Quando il codice viene eseguito con queste precondizioni, il valore di `array1_size` ricercato nella cache causa un cache miss; mentre il processore attende che venga caricato dalla DRAM, inizia ad eseguire speculativamente il branch (siccome l'if verrà predetto come vero).

Nell'esecuzione speculativa si aggiunge `x` all'indirizzo base di `array1` e si richiedono i dati all'indirizzo risultante; la lettura sarà un cache hit, e restituirà il valore del byte segreto k. La logica dell'esecuzione speculativa poi userà k per calcolare l'indirizzo di `array2[k * 4096]`, mandando una richiesta per leggere i dati in memoria a questo indirizzo. Nel frattempo, dopo aver verificato che la condizione dell'if era falsa, il processore ripristinerà lo stato dei registri a prima dell'esecuzione speculativa annullandola. Tuttavia, la lettura speculativa da `array2` influenza lo stato della cache in modo specifico dall'indirizzo, dove l'indirizzo dipende da k.

Per completare l'attacco, l'attaccante misura infine quale locazione in `array2` è stata messa nella cache, e.g. tramite Flush+Reload o Prime+Probe. Questo rivela il valore di k, siccome l'esecuzione speculativa della vittima aveva inserito nella cache `array2[k *4096]`. In modo alternativo, l'attaccante può anche usare Evict+Time, cioè chiamando immediatamente la funzione target di nuovo con un valore *in-bound* `x'` e misurare quando tempo impiega questa seconda chiamata. Se `aray1[x']` equivale a k, allora la locazione acceduta in `array2` è nella cache, e l'operazione tende ad essere più veloce.

Sono stati condotti diversi esperimenti in [12] usando questa variante su più processori, inclusi Intel Ivy Bridge (i7-3630QM) e AMD Ryzen, e tutti hanno riportato la vulnerabilità Spectre. Risultati simili sono stati osservati su architetture a 32 e 64 bit; inoltre, alcuni processori ARM supportano l'esecuzione speculativa; test su Qualcomm Snapdragon 835 SoC (con una CPU Qualcomm Kyro 280) e su un Samsung Exynos 7420 Octa Soc (con CPU

Cortex-A57 e Cortex-A53) hanno confermato che anche questi processori ARM ne sono affetti.

Implementazione usando JavaScript e eBPF È possibile implementare l'attacco Spectre anche in JavaScript e usando l'interfaccia eBPF.

Per quanto riguarda JavaScript, i ricercatori sono riusciti a leggere, attraverso un sito web, la memoria privata del processo del browser in esecuzione. La difficoltà principale è rappresentata dal fatto che l'istruzione `clflush` non è accessibile da JavaScript; una soluzione alternativa è usare la cache eviction: l'idea è quella di accedere altre locazioni di memoria in modo che le locazioni obiettivo siano rimosse successivamente (evicted). Per completare l'attacco, occorre una componente di timing per verificare la velocità di accesso alla linea di cache rimossa; per fare ciò, solitamente si usa l'istruzione `rtdscp`[4]. JavaScript tuttavia non fornisce tale istruzione, e i browser degradano intenzionalmente l'accuratezza dei timer ad alta risoluzione proprio per dissuadere side channel attack basati sulla temporizzazione. Tuttavia, la feature di HTML5 Web Workers[5] rende semplice la creazione di un thread separato che decrementa ripetutamente un valore in una locazione di memoria condivisa. Questo approccio fornisce un timer ad alta risoluzione con una risoluzione sufficiente.

Berkeley Packet Filter (BPF) nacque come semplice linguaggio per scrivere utilities di packet-filtering come `tcpdump`, e fu aggiunto in Linux a partire dal kernel di sviluppo 2.5; solo dalla release 3.0 è stato aggiunto un compilatore just-in-time (JIT) all'interprete BPF e dal kernel 3.4 con il miglioramento del servizio `seccomp`[6] è stato reso possibile per gli utenti fornire un filtro per le system calls scritto nel linguaggio BPF [13].

[4]È un'istruzione che permette di leggere il time-stamp del processore; viene usata appunto per implementare funzionalità di cronometraggio del tempo di esecuzione delle istruzioni.

[5]Si confronti `https://www.w3schools.com/html/html5_webworkers.asp`

[6]Si veda `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`

eBPF (extended BPF) è un'interfaccia del kernel linux basata su BPF che può essere usata come accennato sopra per diversi scopi; eBPF permette ad utenti senza privilegi di innescare l'interpretazione della compilazione JIT e la susseguente esecuzione di bytecode eBPF fornito dall'utente e verificato dal kernel [12]. L'idea dietro l'attacco basato su eBPF è simile a quella per JavaScript.

In questo attacco, si usa codice eBPF solo per il codice eseguito speculativamente, mentre il codice eseguito nello spazio utente serve ad acquisire le informazioni attraverso il canale nascosto, a differenza dell'attacco precedente, dove entrambe le funzioni erano implementate nello stesso linguaggio.

Il sottosistema eBPF gestisce strutture dati salvate nella memoria del kernel. Gli utenti possono richiedere la creazione di queste strutture dati, che possono essere anche accedute dal codice eBPF. Per rinforzare la sicurezza della memoria per queste operazioni, il kernel salva alcuni metadati associati ad ogni struttura dati ed esegue controlli su questi metadati. In particolare, i metadati includono la dimensione della struttura dati (impostata quando la struttura dati è creata usata per prevenire accessi fuori limite) e il numero di riferimenti fatti dai programmi eBPF che sono caricati nel kernel. Il contatore dei riferimenti tiene traccia di quanti programmi eBPF che si riferiscono alla struttura dati sono attualmente in esecuzione, accertandosi che la memoria che appartiene alla struttura dati non sia rilasciata mentre i programmi eBPF caricati la referenziano. Per accedere speculativamente locazioni nello spazio utente, basta eseguire accessi fuori dai limiti dalle strutture dati nel kernel, con un indice abbastanza grande tale da far accedere appunto allo spazio utente. È possibile migliorare il successo dell'attacco attraverso il false sharing[7] poiché è possibile aumentare la latenza dei controlli sulla dimensione delle strutture dati, in quanto il kernel mantiene la lunghezza della struttura dati e del contatore dei riferimenti sulla stessa linea.

[7]Fenomeno che avviene quando diversi thread su diversi processori modificano variabili che risiedono sulla stessa linea di cache; questo invalida la linea di cache e costringe ad un aggiornamento, che degrada le performance. Cfr `https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads`

Mitigazioni

Diverse contromisure sono state proposte per contrastare una o più delle caratteristiche su cui si basa l'attacco; ne sono qui proposte alcune.

1. **Prevenire l'esecuzione speculativa**: con questa soluzione, gli attacchi Spectre non sarebbero più possibili; tuttavia, ciò comporterebbe un enorme degrado delle perfomance. Una soluzione protrebbe essere quella di permettere la disabilitazione dell'esecuzione speculativa tramite software, e.g. con istruzioni che blocchino l'esecuzione speculativa e che assicurino che le istruzioni nel blocco non siano eseguite speculativamente, una caratteristica non presente negli attuali processori.

   Un'altra soluzione potrebbe essere quella dell'utilizzo dell'istruzione `lfence` prima di un branch, che permette di completare tutte le istruzioni precedenti a `lfence`, in modo che il branch sia eseguito velocemente e non ci siano altre esecuzioni speculative che ne possano ritardare la risoluzione. Questa soluzione richiede però che tutto il software legacy venga aggiornato.

2. **Prevenire l'accesso ai dati segreti**: altre contromisure possono impedire al codice eseguito speculativamente di accedere a dati segreti. Una di queste misure, usata dal browser Google Chrome, è di eseguire ogni sito web in un processo separato. Siccome gli attacchi Spectre influenzano solo i permessi della vittima, un attacco come quello con JavaScript non sarebbe in grado di accedere dati dai processi assegnati ad altri siti web. Anche webkit adotta alcune strategie: una di esse è quella di non controllare se l'indice di un array è nei valori limite, ma piuttosto di effettuare una operazione sui bit dell'indice (applicando una *maschera*) assicurando così che l'indice non sia troppo più grande della dimensione dell'array. Questo non impedisce di accedere fuori dai limiti dell'array, ma previene l'accesso a locazioni arbitrarie di memoria.

3. **Limitare l'esfiltrazione di dati dai covert channel**: rimuovere la possibilità di recuperare i dati ottenuti con l'esecuzione speculativa è un altro metodo di mitigazione. Questo è possibile limitando la risoluzione dei timer di JavaScript, aggiungendo eventualmente del disturbo. Non è chiaro però il livello di protezione che fornirebbe questa soluzione siccome le fonti di errore riducono solo il rateo al quale gli attaccanti possono recuperare i dati. Inoltre, i processori correnti deficitano dei meccanismi richiesti per l'eliminazione completa dei canali nascosti [12]. Quindi, mentre questo approccio potrebbe ridurre le performance dell'attacco, non garantisce di eliminarlo completamente.

2.4 Attacchi basati sull'esecuzione fuori ordine

L'esecuzione fuori ordine è un'altra tecnica di ottimizzazione che permette di massimizzare l'utilizzo dei core della CPU. Consiste nell'eseguire certe sequenze di istruzioni prima che quelle precedenti siano state completate. È doveroso a questo punto osservare quali sono le differenze e le similitudini tra la BPU, l'esecuzione speculativa e l'esecuzione fuori ordine: insiemi di istruzioni possono essere eseguiti speculativamente (ovvero, senza che il processore sia certo che debbano davvero essere eseguite) ma in ordine; si può anche avere una situazione in cui le istruzioni non sono eseguite speculativamente, ma sono riordinate in qualche modo dal processore; infine il predittore dei salti permette di stabilire se un ramo debba essere preso o meno, o se più in generale un'istruzione è un'istruzione di salto. Questi tre strumenti in sostanza sono indipendenti tra loro, anche se strettamente correlati ed usati insieme. Per i dettagli sull'esecuzione fuori ordine si può consultare la sezione 1.5.

Anche questa tecnica può essere sfruttata per rubare informazioni riservate: infatti, l'esecuzione fuori sequenza è in grado di modificare lo stato microarchitetturale in modo tale da lasciare delle informazioni che possono

essere carpite da un avversario; l'attacco Meltdown [14] è proprio in grado di fare ciò.

2.4.1 Meltdown

Consideriamo il seguente listato:

```
raise_exception();
//la linea sottostante non è mai raggiunta
access(probe_array[data*4096]);
```

Il frammento solleva un'eccezione non gestita e poi accede ad un array. Quando si solleva un'eccezione, il flusso di controllo non segue mai il codice dopo l'eccezione, ma salta ad un gestore delle eccezioni nel sistema operativo. Quindi il flusso di controllo continua nel kernel e non più con la prossima istruzione nello spazio utente.

Nell'esempio mostrato quindi in teoria non è mai possibile accedere all'array, siccome l'eccezione passa il controllo al kernel e l'applicazione termina. Tuttavia, grazie all'esecuzione fuori ordine, la CPU potrebbe già aver eseguito le istruzioni seguenti siccome non c'è alcuna dipendenza sull'istruzione precedente che innesca l'eccezione. A causa dell'eccezione, poi, le istruzioni eseguite fuori ordine non sono ritirate e quindi non hanno effetti architetturali visibili sui registri o sulla memoria.

Nonostante non ci siano effetti diretti architetturali, ci sono però effetti collaterali microarchitetturali: durante l'esecuzione fuori ordine, la memoria referenziata è caricata in un registro e salvata anche nella cache. Se l'esecuzione fuori ordine deve essere scartata, non si fa il commit dei contenuti del registro e della memoria; tuttavia, i contenuti della cache sono ivi mantenuti. Con un attacco microarchitetturale side channel come Flush+Reload, che determina se una locazione di memoria specifica è nella cache, è possibile rendere visibile (per un attaccante) questo stato microarchitetturale.

Anche altri attacchi (Prime+Probe, Evict+Reload) possono essere impiegati, ma Flush+Reload è il più accurato ed è semplice da implementare [14].

In base al valore di `data` in esempio, quando si esegue l'accesso di memoria fuori ordine è acceduta una diversa parte della cache. Come `data` viene moltiplicato per 4096, gli accessi ai dati di `probe_array` sono sparsi sull'array con una distanza di 4KB (supponendo che il tipo di dato per `probe_array` abbia dimensione 1B). Quindi, siccome c'è una corrispondenza iniettiva che va dal valore di `data` ad una pagina di memoria, i.e. valori diversi per i dati non risultano mai in un accesso alla stessa pagina, di conseguenza se una linea di cache di una pagina è messa nella cache, sappiamo il valore di `data`.

Anche se l'accesso all'array non deve avvenire a causa dell'eccezione, l'indice che sarebbe stato acceduto viene messo nella cache. Vediamo ora più nel dettaglio come è costituito l'attacco.

Descrizione dell'attacco

L'attacco si divide in 3 passi:

1. Viene caricato in un registro il contenuto di una locazione di memoria scelta dall'attaccante, inaccessibile a quest'ultimo.

2. Un'**istruzione transiente** accede ad una linea di cache in base al contenuto del registro.

3. L'attaccante usa Flush+Reload per determinare la linea di cache acceduta e di conseguenza il segreto "custodito" (memorizzato) alla locazione di memoria scelta.

Come visto per Spectre, un'istruzione transiente è un'istruzione che non si dovrebbe incontrare mai nel flusso d'esecuzione (nell'esempio della sezione precedente era l'accesso all'array). Queste istruzioni vengono appunto eseguite fuori ordine e lasciano un effetto collaterale misurabile. Inoltre, una qualsiasi sequenza di istruzioni contenente almeno un'istruzione transiente è detta sequenza di istruzioni transienti.

Ripetendo i passi appena illustrati, un attaccante è in grado di copiare (*dump*) l'intera memoria del kernel, inclusa tutta la memoria fisica.

Passo 1: Leggere il segreto Per caricare dati dalla memoria principale ad un registro, questi devono essere referenziati usando un indirizzo virtuale. Oltre a tradurre un indirizzo virtuale in un indirizzo fisico, la CPU controlla anche i bit dei permessi dell'indirizzo virtuale, i.e. se questo indirizzo virtuale è accessibile dall'utente o dal kernel. Questo isolamento basato sull'hardware attraverso un *permission bit* è considerato sicuro; quindi i sistemi operativi moderni indirizzano l'intero kernel nello spazio di indirizzamento virtuale di ogni processo utente. Di conseguenza, tutti gli indirizzi del kernel conducono ad un indirizzo fisico valido quando tradotti, e la CPU può accedere i contenuti di quegli indirizzi. L'unica differenza rispetto ad accedere un indirizzo dello spazio utente è che la CPU solleva un'eccezione poiché il livello corrente dei permessi non consente di accedere un tale indirizzo[8]. Quindi, lo spazio utente non può semplicemente leggere i contenuti di un tale indirizzo. Tuttavia, Meltdown sfrutta l'esecuzione fuori ordine delle CPU moderne,

[8]Per isolare i processi tra loro, la CPU supporta spazi di indirizzamento virtuali dove indirizzi virtuali sono tradotti in indirizzi fisici. Uno spazio di indirizzamento virtuale è diviso in un insieme di pagine che possono essere associate a indirizzi della memoria fisica attraverso una tabella di traduzione delle pagine a più livelli (*multi-level page translation table*). La tabella di traduzione definisce la corrispondenza effettiva tra indirizzi fisici e virtuali e proprietà di protezione che sono usate per applicare controlli dei privilegi (lettura, scrittura, esecuzione). La tabella correntemente in uso è mantenuta in un registro speciale della CPU. Ad ogni context switch, il sistema operativo aggiorna questo registro con la tabella del prossimo processo, al fine di implementare spazi di indirizzi virtuali per ogni processo. Ogni processo di conseguenza può far riferimento a dati che appartengono al proprio spazio di indirizzamento virtuale. Ogni spazio è diviso in una parte utente e una parte kernel. Mentre lo spazio utente può essere acceduto dall'applicazione in esecuzione, lo spazio di indirizzi del kernel può essere acceduto solo se la CPU è in modalità privilegiata. Il sistema operativo applica ciò disabilitando la proprietà che garantisce l'accessibilità all'utente delle corrispondenti tabelle di traduzione. Lo spazio di indirizzi del kernel non ha solo memoria mappata per l'uso del kernel, ma siccome deve eseguire anche operazioni sulle pagine utente, l'intera memoria fisica è tipicamente mappata nel kernel.

che esegue comunque istruzioni nella breve finestra di tempo tra l'accesso di memoria illegale e il sollevamento dell'eccezione.

Consideriamo il seguente codice:

```
1 ; rcx = indirizzo del kernel, rbx = array di sonda
2 xor rax, rax
3 retry:
4 mov al, byte[rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2.2: il nucleo dell'attacco.

alla linea 4, il valore del byte localizzato dall'indirizzo del kernel obiettivo, che è salvato nel registro `RCX`, è caricato nel byte meno significativo del registro `RAX` rappresentato da `AL`. L'operazione `MOV` non è atomica, ma è divisa in sotto-operazioni (μOPs); cercando di utilizzare la pipeline il più possibile, la CPU inizia l'esecuzione delle istruzioni successive (5-7). Le micro operazioni vengono poi "ritirate" in ordine; durante questa fase, viene gestita qualsiasi eccezione o interrupt avvenuto durante l'esecuzione delle istruzioni e di conseguenza la pipeline viene "scaricata" (*flushed*) e vengono eliminati tutti i risultati dati dall'esecuzione fuori ordine di istruzioni successive a quella dell'eccezione. Tuttavia, c'è una *race condition* tra il sollevamento dell'eccezione e lo step 2 dell'attacco, descritto qua sotto.

Passo 2: Trasmettere il segreto La sequenza di istruzioni dal passo 1 che viene eseguita fuori sequenza deve essere scelta in modo che che diventi una sequenza di istruzioni *di transizione*. Se questa sequenza è eseguita prima che l'istruzione `MOV` sia ritirata (i.e. solleva l'eccezione), e la sequenza di istruzioni di transizione effettua computazioni basate sul segreto, può essere utilizzata per trasmettere il segreto all'attaccante. Come precedentemente discusso, si è scelto di usare la cache come covert channel (e Flush+Reload per recuperare i dati, ma è possibile usare altri elementi architetturali per

creare un canale nascosto). Quindi, la sequenza di istruzioni transienti deve codificare il segreto nello stato microarchitetturale della cache.

Per fare ciò, si alloca un array di sonda (*probe array*) in memoria; bisogna assicurare che nessuna parte dell'array sia nella cache. Per trasmettere il segreto, la sequenza di istruzioni transienti deve contenere un accesso indiretto ad un indirizzo che viene calcolato in base al valore segreto (inaccessibile). Nella riga 5 del listato 2.2, il valore segreto del passo 1 è moltiplicato per la dimensione della pagina (4KB). La moltiplicazione del segreto assicura che gli accessi all'array abbiano una larga distanza spaziale l'uno con l'altro. Questo previene il prefetcher hardware dal caricare locazioni di memoria adiacenti nella cache. In questo caso leggiamo un singolo byte alla volta; quindi l'array sonda sarà 256×4096 bytes, assumendo pagine da 4KB.

Nella linea 7 il segreto moltiplicato è aggiunto all'indirizzo base dell'array sonda, formando l'indirizzo target del canale nascosto. Questo indirizzo è letto per inserire nella cache la linea di cache corrispondente. L'indirizzo sarà caricato nella cache L1 del core richiedente e, grazie all'inclusività, anche nella cache L3 dove può essere letto da altri core.

Siccome la sequenza di istruzioni transienti nello step 2 "corre" contro il sollevamento dell'eccezione, ridurre il tempo di esecuzione dello step 2 può migliorare le performance dell'attacco.

Passo 3: Ricevere il segreto In questo step l'attaccante recupera il valore segreto con un attacco side channel microarchitetturale che trasferisce lo stato della cache (step 2) in uno stato architetturale (e.g. con Flush+Reload). Quando la sequenza di istruzioni temporanee dello step 2 è eseguita, esattamente una linea della cache dell'array sonda è nella cache. La posizione della linea della cache all'interno dell'array sonda dipende solo dal segreto che è letto allo step 1. Quindi l'attaccante itera attraverso tutte le 256 pagine e misura il tempo di accesso per ogni offset nella pagina. Il numero della pagina contenente la linea di cache memorizzata corrisponde direttamente al valore segreto.

Ripetendo tutti questi 3 passi, un attaccante può copiare l'intera memoria iterando su tutti gli indirizzi. Siccome inoltre tutti i principali sistemi operativi mappano l'intera memoria fisica nello spazio di indirizzi del kernel in ogni processo utente, Meltdown può anche copiare l'intera memoria fisica della macchina sotto attacco.

Sistemi affetti da Meltdown

Meltdown è stato testato con successo su Linux, Windows e Android. I processori AMD sembrano però essere immuni al bug [16].

È interessante analizzare sopratutto il suo impatto su ambienti (semi) virtualizzati: infatti Meltdown può essere usato contro container come Docker, ottenendo informazioni non solo dal kernel sottostante, ma anche dagli altri container eseguiti sullo stesso host fisico; questo perché ogni container usa lo stesso kernel, che è appunto condiviso tra tutti i container. Di conseguenza, Meltdown affligge pesantemente i *cloud providers*, sopratutto se i *guest* non sono completamente virtualizzati: per motivi di performance, molti provider di servizi di hosting o di cloud non dispongono di un livello di astrazione per la memoria virtuale; in tali ambienti il kernel è appunto condiviso tra tutti gli ospiti. Quindi l'isolamento tra gli utenti può semplicemente essere aggirato con Meltdown, esponendo completamente i dati di tutti gli altri ospiti sull'host. Per questi provider, cambiare l'infrastruttura per virtualizzare completamente o usare soluzioni software come KAISER (cfr 2.7.1) farebbe aumentare i costi in modo significativo.

La situazione è particolarmente grave se si pensa che la richiesta e l'utilizzo dei servizi cloud è in costante aumento, come si legge ad esempio da un documento dell'eurostat di dicembre 2018 [15].

Contromisure

Hardware Meltdown bypassa il l'isolamento tra domini di sicurezza rinforzato dall'hardware. Non c'è alcuna vulnerabilità software coinvolta; qualsiasi patch software lascerà piccole quantità di memoria esposta. Non c'è

alcuna documentazione riguardo a se una soluzione richieda hardware completamente nuovo oppure possa bastare un aggiornamento del microcodice. Una contromisura banale naturalmente è quella di disabilitare l'esecuzione fuori ordine; ma questo sarebbe inaccettabile per via della perdita di performance. Un'altra via applicabile è quella di introdurre una separazione forte tra lo spazio utente e lo spazio kernel. Questo potrebbe essere abilitato opzionalmente nei nuovi kernel usando un bit in un registro di controllo della CPU; se il bit è impostato, il kernel deve risiedere nella metà superiore dello spazio degli indirizzi, e lo spazio utente deve risiedere nella metà inferiore dello spazio degli indirizzi. Un fetch di una locazione di memoria può in questo modo essere immediatamente identificato, ovvero se l'accesso viola il confine di sicurezza, poiché il livello di autorizzazione può essere derivato direttamente dall'indirizzo virtuale senza nessun'altra ricerca. L'impatto sulle performance previsto sarebbe minimo [14]. In più, sarebbe garantita la retrocompatibilità, siccome il bit di divisione sarebbe abilitato solo se l'hardware supporta la feature.

Bisogna notare che queste contromisure però prevengono solo Meltdown, e non Spectre, così come le contromisure per Spectre non affliggono Meltdown.

KAISER KAISER è una modifica del kernel proposta per non avere lo spazio del kernel mappato nello spazio utente. KAISER serve per prevenire attacchi side channel contro KASLR (cfr sezione 2.7.1 per i dettagli). Tuttavia, previene anche Meltdown, in quanto assicura che non ci sia una corrispondenza valida allo spazio kernel o alla memoria fisica disponibile nello spazio utente. KAISER ha comunque qualche limitazione: infatti, a causa del design dell'architettura x86, è comunque richiesto che diverse locazioni di memoria privilegiate (kernel) siano mappate nello spazio utente, ovvero queste locazioni di memoria possono lo stesso essere lette dallo spazio utente. Anche se queste locazioni di memoria non contengono alcuni segreti (e.g. credenziali), possono comunque contenere puntatori. Se anche un solo puntatore viene scoperto, è possibile aggirare KASLR, in quanto la rando-

mizzazione può essere calcolata dal valore del puntatore. KAISER rimane comunque la miglior soluzione immediata attualmente disponibile

2.4.2 Note finali su Spectre e Meltdown

Meltdown è distinto da Spectre in due modi principali: anzitutto, a differenza di Spectre, non usa la predizione dei salti. Si basa invece sull'osservazione che quando un'istruzione causa una trap, le istruzioni successive sono eseguite tramite il meccanismo fuori sequenza prima che il processo sia terminato. Secondariamente, Meltdown sfrutta una vulnerabilità specifica a diversi processori Intel e alcuni processori ARM che permettono ad alcune istruzioni eseguite speculativamente di bypassare la protezione della memoria. Combinando questi problemi, Meltdown è in grado di accedere la memoria del kernel dallo spazio utente. Questo accesso causa una trap, ma prima che sia gestita, le istruzioni che seguono l'accesso leggono i contenuti della memoria acceduta attraverso un canale nascosto.

Gli attacchi Spectre invece lavorano su una gamma più vasta di processori, inclusi molti processori AMD e ARM. In più, il meccanismo KAISER, che è in grado di mitigare Meltdown, non ha alcun potere contro Spectre.

2.5 Attacchi basati sulla DRAM

Nel 2014, Kim et al. [17] studiarono gli effetti dei cosiddetti rumori di disturbo (*disturbance error*) su chip off-the-shelf DRAM: i ricercatori mostrarono che leggendo continuamente lo stesso indirizzo nella DRAM è possibile corrompere i dati negli indirizzi vicini; più nello specifico, attivare la stessa riga nella DRAM corrompe i dati nelle righe vicine. Questo fenomeno fu chiamato RowHammer bug, e aprì sostanzialmente un nuovo campo di ricerca. RowHammer è inoltre probabilmente il primo esempio di come un meccanismo di fallimento a livello dei circuiti può essere sfruttato in modo pratico come una vulnerabilità della sicurezza di un sistema, tramite software [6], [18]. Questo attacco infatti si differenzia dai precedenti in quanto

induce un guasto hardware, cosa che gli attacchi precedenti non facevano (è comunque possibile usare la DRAM per attacchi che non inducono guasti, come DRAMA [6], [19]).

Sono seguiti appunto molti studi che hanno specializzato l'attacco originario, ponendo molte minacce per la sicurezza di diversi sistemi (computer, cloud, smartphone), illustrati nelle sezioni successive.

2.5.1 RowHammer bug

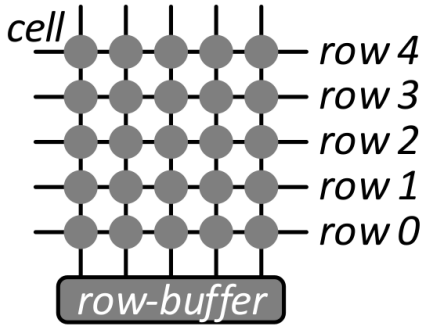

Figura 2.1: Organizzazione della DRAM.

Organizzazione della DRAM Un chip DRAM contiene varie *celle*, ognuna delle quali è composta da un *condensatore* e un *transistor* di accesso. A seconda di se il transistor sia completamente carico o scarico, una cella è nello *stato carico* o nello *stato scarico*, rispettivamente. Questi due stati sono usati per rappresentare un valore binario.

Ogni cella giace all'intersezione di due fili perpendicolari: quello orizzontale è detto *wordline* e quello verticale è detto *bitline*. Il primo collega tutte le celle nella direzione orizzontale, formando una *riga*, mentre il secondo nella direzione verticale, formando una *colonna*. Quando si aumenta il voltaggio di una *wordline*, si abilitano tutti i transistor di accesso nella *riga*, che collegano tutti i condensatori alle *bitline* rispettive. Questo permette ai dati (sotto

forma di cariche) nella riga di essere trasferiti al *row-buffer*. Il row-buffer legge le cariche dalle celle, un processo che distrugge i dati in esse, ma le riscrive immediatamente nelle stesse. Quando non ci sono più accessi alla riga il voltaggio nella wordline è abbassato, disconnettendo i condensatori dalle bitline. Un gruppo di righe è chiamato *banco* (bank), ognuno dei quali ha il proprio row-buffer. Più banchi insieme formano un *rank*.

Per accedere ad un rank occorre anzitutto aprire una riga: per farlo si alza una wordline. Questo collega la riga alle bitline, trasferendo tutti i dati nel row-buffer del banco. Successivamente, i dati nel row-buffer sono acceduti leggendo o scrivendo le sue colonne; infine, si deve chiudere la riga, abbassando la wordline. Prima di aprire una nuova linea, bisogna chiudere quella attualmente aperta.

La carica in una cella DRAM non è persistente. Ciò è dovuto a vari meccanismi a causa dei quali la carica si può disperdere; col tempo, il livello di carica della cella può essere deviato oltre la soglia di rumore, causando la perdita di dati; in altre parole, una cella ha un *tempo di ritenzione* limitato. Prima che questo tempo scada, la carica della cella deve essere ripristinata (cioè bisogna fare un *refresh*) al suo valore originale: completamente carica o completamente scarica. Le specifiche della DRAM DDR3 garantiscono un tempo di ritenzione di almeno 64 millisecondi, il che implica che tutte le celle all'interno del rank devono essere ricaricate almeno una volta entro questa finestra temporale. Ricaricare una cella può essere fatto aprendo la riga alla quale la cella appartiene. Da un punto di vista del circuito quindi ricaricare una cella e aprire una cella sono operazioni identiche.

Errori di disturbo In generale, gli errori di disturbo avvengono ogni qual volta c'è una interazione sufficientemente forte tra due componenti di un circuito (e.g. condensatori, transistor, fili) che dovrebbero essere isolati tra loro. A seconda di quali componenti, e come interagiscono, sono possibili diverse modalità di disturbo differenti. In particolare, una modalità di disturbo affligge i chip DRAM: quando la tensione di una wordline è cambiata

ripetutamente, alcune celle nelle righe vicine perdono la carica ad una velocità molto più veloce. Queste celle non possono trattenere la carica per 64 ms, l'intervallo di tempo al quale sono ricaricate. Questo può portare le celle a perdere dati e ad avere errori di disturbo. I ricercatori hanno avanzato diverse ipotesi su quali possano essere le cause [17], e hanno scoperto che questo problema era molto diffuso (hanno trovato 110 moduli che soffrivano del bug RowHammer su 129 testati).

Nell'articolo originale, i ricercatori hanno usato un test per individuare le celle che sono disturbate dopo aver "martellato" ogni riga più volte (ovvero attivato ripetutamente ogni riga, da cui il nome dell'attacco). Un elemento fondamentale per il successo dell'attacco è lo svuotamento della cache. Infatti, siccome bisogna accedere più volte alla stessa riga, affinché l'accesso avvenga realmente bisogna che i dati siano rimossi dalla cache; nell'articolo originale si era usato il comando `clflush`, ma sono state studiate anche altre tecniche, anche per permettere l'attacco da ambienti privi del comando (e.g. JavaScript) [20].

Dopo questa scoperta, numerosi studi sono stati effettuati sulle possibilità che questo attacco offre; questi studi sono condensati in [18] e nella sezione 2.5.2 è fornito un esempio.

La soluzione PARA Sono state proposte alcune soluzioni; nell'articolo originale, la migliore in termini dei costi (prestazione, difficoltà di implementazione...) è denominata **PARA**, Probabilistic Adjacent Row Activation. L'idea chiave è semplice: tutte le volte che una riga è aperta e chiusa, viene aperta anche una delle righe adiacenti (cioè ricaricata) con una qualche (bassa) probabilità. Se una riga particolare viene aperta e chiusa ripetutamente, allora è statisticamente sicuro che le righe adiacenti saranno eventualmente aperte. Il vantaggio principale di PARA è l'essere *stateless*: non richiede strutture dati hardware costose per contare il numero di volte che le righe sono state aperte o di memorizzare l'indirizzo delle righe vittime/attaccanti.

PARA è implementato nel controller della memoria; quando una riga è

chiusa, il controller lancia una moneta sbilanciata, con una probabilità p di uscire testa, dove $p \ll 1$. Se esce testa, il controllore apre una delle due righe adiacenti dove entrambe le righe sono scelte con probabilità uguale ($p/2$). A causa della natura probabilistica, PARA non garantisce che le righe adiacenti siano sempre ricaricate in tempo, quindi PARA non può prevenire errori di disturbo con certezza assoluta. Tuttavia, il parametro p può essere impostato in modo che questi errori avvengano con una probabilità estremamente bassa - molti ordini di grandezza inferiori rispetto al rateo di fallimento di altri componenti di sistema (e.g. più dell'1% dei guasti agli hard disk ogni anno[17]).

2.5.2 Drammer

Drammer, ovvero Deterministic Rowhammer Attacks on Mobile Platforms [21] è uno degli attacchi che sfrutta il RowHammer bug. Questo attacco, nel 2016, è stato il primo a dimostrare l'applicabilità di RowHammer al di fuori delle architetture x86, nello specifico su architetture ARM, dimostrando quindi che anche gli smartphone possono soffrire della vulnerabilità. L'attacco creato inoltre non richiede nessun permesso utente e non sfrutta alcuna vulnerabilità software. Infine, è stato il primo attacco deterministico (e non probabilistico, come i precedenti) e il primo che non richiede caratteristiche speciali di gestione della memoria come la deduplicazione della memoria[9]. La tecnica introdotta per ottenere questi risultati è detta **Phys Feng Shui**, in quanto è un'istanza di *Flip Feng Shui* (FFS), una tecnica di exploitation per indurre, come RowHammer, cambiamenti nei bit di dati sensibili scelti dall'attaccante. Lo scopo di questo attacco è ottenere il permesso di root. L'unica premessa dell'attacco, è che l'attaccante abbia a disposizione un'app che possa controllare installata sul telefono della vittima.

[9]Tecnica che permette di unire pagine di memoria con gli stessi contenuti, permettendo di conseguenza un risparmio della memoria utilizzata.

Descrizione dell'attacco

È importante notare anzitutto che innescare il bug RowHammer è differente dall'usarlo (i.e. sfruttarlo) in modo relativo alla sicurezza del sistema. Un exploit deve convincere (con l'inganno) una componente della vittima (e.g. un altro processo, il sistema operativo, etc.) ad usare una locazione di memoria fisica vulnerabile per salvare dei contenuti sensibili.

Nel bug RowHammer, innescare inversioni dei bit è sostanzialmente una corsa contro il *refresh* della DRAM mentre si cercano di effettuare un numero sufficiente di accessi ad una riga per disturbare quelle adiacenti. L'attacco *single-sided* si basa solamente su una riga "aggreditrice" per attaccarne una adiacente, mentre nel più efficiente *double-sided RowHammer* si accedono le due righe che sono direttamente sopra o sotto la riga della vittima.

In ogni attacco che sfrutta RowHammer ci sono 3 primitive che devono essere soddisfatte:

1. **Accessi veloci alla memoria non in cache**: la memoria non deve essere nella cache per poter essere ovviamente acceduta direttamente dalla DRAM; oltre a dover svuotare la cache, l'altro aspetto non banale è che questi accessi devono essere *veloci*, ovvero avvenire entro la finestra di tempo di refresh (che per le DDR3 è 64 ms).

2. **Massaggio della memoria fisica**: la vittima deve usare una cella di memoria che è soggetta al bug RowHammer. L'attaccante deve quindi essere in grado di *massaggiare* la memoria in modo abbastanza preciso per ingannare la vittima e spingerla a usare la cella vulnerabile per salvarci informazioni sensibili. La parte più impegnativa è riuscire a implementare la primitiva in modo completamente *deterministico*.

3. **Indirizzamento della memoria fisica**: per accedere la memoria da due righe attaccanti, un attaccante deve sapere quali indirizzi virtuali corrispondono agli indirizzi fisici delle righe.

Implementazione

Un prerequisito per implementare primitive efficaci è quello di comprendere il modello della memoria del chip sotto attacco. Una delle proprietà chiave da determinare è la *dimensione della riga*. Un modo per farlo è con un side channel basato sul tempo (*timing-base side channel*): l'idea è che accedere a due pagine di memoria dallo stesso banco è più lento che leggerle da banchi differenti: per accessi dallo stesso banco, il controller deve ricaricare il row buffer del banco per ogni operazione di lettura.

Le piattaforme di computazione moderne consistono di diversi componenti hardware differenti: il SoC (System-on-Chip, comprende la CPU), si trovano spesso una GPU, un controller del display, una camera, etc. Per supportare efficientemente la condivisione della memoria tra questi elementi, così come tra i servizi a livello utente, il sistema operativo deve fornire meccanismi di accesso diretto alla memoria, **DMA** (Direct Memory Access). Siccome processando pipeline che coinvolgono buffer della DMA si aggirano la CPU e le sue cache, il sistema operativo deve agevolare la gestione esplicita della cache per far sì che ogni componente della pipeline abbia una visione coerente della memoria sottostante. Inoltre, siccome la maggior parte dei dispositivi effettuano operazioni DMA solamente a pagine di memoria fisicamente contigue, il sistema operativo deve fornire anche allocatori che supportano questo tipo di memoria. Chiamiamo l'interfaccia del sistema operativo che fornisce tutti questi meccanismi *DMA buffer management API*. Per costruzione, i buffer DMA accessibili dagli utenti implementano due delle primitive d'attacco: **(1) fornire accessi della memoria uncached** e **(3) indirizzamento della memoria fisica**.

Per la primitiva rimanente **(2)**, bisogna disporre la memoria fisica in modo tale che è possibile controllare il contenuto di una pagina di memoria vulnerabile e inserirci *deterministicamente* informazioni sensibili per la sicurezza. A questo scopo, in [21] è proposto Phys Feng Shui, una tecnica per effettuare il *massaggiamento* della memoria basato solamente su pattern predicibili di riutilizzo della memoria degli allocatori della memoria fisica. Questa tecni-

ca inoltre non incappa nel rischio di far crashare accidentalmente il sistema causando inversioni di bit in parti non intenzionali di memoria fisica.

Bisogna quindi anzitutto effettuare un processo di *memory templating*, ovvero scoprire quali locazioni di memoria sono suscettibili a RowHammer[10]. Una sessione di templating di successo fornisce una lista di template che contengon la locazione di bit vulnerabili, insieme alla direzione dell'inversione, ovvero 0-to-1 oppure 1-to-0.

Le piattaforme Linux gestiscono la memoria fisica attraverso l'allocatore *buddy* [21], il cui scopo è minimizzare la frammentazione esterna dividendo e unendo efficientemente la memoria disponibile in blocchi la cui dimensione è una potenza di 2. Ad ogni richiesta di deallocazione, quindi, l'allocatore esamina blocchi vicini che hanno la stessa dimensione per unirli se sono liberi. Per minimizzare la frammentazione interna prodotta dall'allocatore per i piccoli oggetti, Linux implementa un'astrazione sopra di esso tramite un "allocatore slab" (a lastre). L'implementazione di default dell'allocatore *SLUB* organizza piccoli oggetti in un numero di pool (o *slab*) di dimensioni comunemente usate per servire velocemente richieste di allocazione e deallocazione. Ogni slab è espando on demand di una dimensione predeterminata per-slab allocata attraverso l'allocatore buddy.

Phys Feng Shui spinge l'allocatore buddy a fargli riusare e partizionare la memoria in modo predicibile. A questo scopo, i ricercatori hanno usato 3 tipi diversi di chunk contigui di memoria: chunk piccoli, detti S, di dimensione fissa di 4KB; chunk medi, M, di dimensione pari a quella di una riga; chunk grandi, L, impostati alla dimensione più grande possibile di chunk contigui di memoria che l'allocatore può fornire.

Preparazione e templating Per prima cosa bisogna *esaurire*, i.e. allocare tutta la memoria fisica contigua disponibile di dimensione L e testarla per

[10]Si veda K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In Proceedings of the 25th USENIX Security Symposium, 2016.

template vulnerabili che possono essere sfruttati più tardi. Dopodiché si esauriscono tutti i chunk di dimensione M.

Riutilizzo selettivo della memoria Si seleziona uno dei template generati al passo precedente come target dell'exploit; il blocco corrispondente (di dimensione L) sarà chiamato **L***. Si rilascia poi L*, e si ri-esauriscono immediatamente di nuovo tutti i blocchi di dimensione M. Siccome erano stati già esauriti al passo precedente, questo costringe l'allocatore a metterli nella regione appena rilasciata, ovvero a riusare in modo predicibile la regione di memoria del chunk appena rilasciato L*. Chiamiamo il chunk M che ora contiene il template **M***.

Infine, in preparazione dell'arrivo della tabella delle pagine (PT) nella pagina vulnerabile di M*, si rilascia M*.

Inserire la prima tabella delle pagine nella regione vulnerabile Si spinge ora l'allocatore della memoria a inserire un chunk S nel chunk vulnerabile M* appena rilasciato. A questo scopo, si svuota l'allocatore dei blocchi disponibili di dimensione S..M/2 allocando ripetutamente chunk di dimensione S. Questo garantisce che le allocazioni S successive finiscano in M*. Si allocano chunk S forzando allocazioni di tabelle delle pagine da 4KB: si mappa ripetutamente la memoria ad indirizzi virtuali fissati che segnano i confini della tabella delle pagine (i.e. ogni 2MB dello spazio di indirizzi virtuali in ARMv7). Siccome il numero massimo di tabelle delle pagine per ogni processo è 1024, si creano un numero di processi per allocarne quante ne servono. Una volta che tutti i chunk più piccoli sono esauriti, la prossima allocazione S finisce prevedibilmente nella regione vulnerabile (non sono disponibili altri blocchi più piccoli).

Determinare quando le allocazioni raggiungono la regione vulnerabile è banale su Linux: il filesystem proc fornisce `/proc/zoneinfo` e `proc/pagetypeinfo`, due file speciali leggibili globalmente che forniscono informazioni sul numero delle pagine di memoria disponibili e allocate, e sulle zone. In caso non siano disponibili, si può sfruttare la Perfomance Monitor Unit per identificare

quando S finisce in M*: infatti, se l'allocatore non può servire una richiesta da un pool di chunk della dimensione richiesta, ma deve dividere un blocco più grande, allora l'operazione richiederà più tempo.

Allineare la tabella delle pagine della vittima Infine, si può mappare una pagina p nel precedente blocco L* che confina con M* a sinistra (nel caso di un'inversione 0-to-1) o a destra (1-to-0), alla locazione fissata nello spazio di indirizzi di memoria virtuale per forzare una nuova allocazione PTP (page table page). A seconda dell'indirizzo virtuale preso, la page table entry (PTE) che punta a p si trova ad un offset diverso nella PTP - permettendo essenzialmente di allineare la PTE della vittima secondo il template vulnerabile.

In modo simile, si può allineare la PTP della vittima in base alla pagina vulnerabile per assicurare di poter invertire i bit selezionati nella PTE della vittima. A questo scopo, si forza l'assegnazione di un numero di PTP di riempimento a seconda delle necessità prima o dopo l'allocazione del PTP della vittima.

Occorre poi che la PTP vulnerabile allocata in M* e la locazione di p siano 2^n pagine distanti: invertire l'n-esimo bit più basso dell'indirizzo fisico della pagina nel PTE della vittima cambia in modo deterministico il PTE per puntare alla PTP vulnerabile stessa, mappando quest'ultima nel nostro spazio di indirizzi. Per ottenere questo, si seleziona qualsiasi pagina p nel chunk M adiacente a M* da mappare nella PTP vittima.

Exploitation Una volta selezionata ed allineata la PTP vittima, la PTE e n a seconda del template vulnerabile, si esegue un RowHammer double-sided e si replica l'inversione del bit trovato nella fase di templating. Attivata l'inversione desiderata, si ottiene l'accesso di scrittura nella pagina essendo ora mappata nello spazio di indirizzi dell'attaccante. È possibile modificare una delle proprie PTP e ottenere l'accesso a qualsiasi pagina nella memoria fisica, inclusa la memoria del kernel.

Notare che un'inversione di bit in una PTE è sfruttabile solo se avviene in uno dei bit inferiori della parte indirizzo: infatti, data una word di 32 bit - ovvero una potenziale PTE - gli offset 1-12 sono parte del campo delle proprietà (e.g. `____ ____ __01 1000 1101|ppro pert iess`). Non funziona nemmeno se il bit invertibile è nella seconda metà di una pagina, in quanto il nuovo indirizzo punterebbe ad una pagina fisicamente 2 GB a destra del suo PTP. Senza l'accesso agli indirizzi fisici assoluti, è possibile supportare solo inversioni di bit che innescano uno shift nell'offset della pagina di al più L-1.

Root privilege escalation Una volta ottenuto il controllo su una delle proprie PTP, è possibile eseguire la parte finale dell'attacco, i.e. ottenere l'accesso di root aumentando i privilegi. Per fare ciò, si mappano ripetutamente diverse pagine fisiche per sondare la memoria del kernel per il contesto di sicurezza del proprio processo (`struct cred`) che si può identificare usando una firma univoca a 24 byte basata su un UID univoco per app.

Nel caso peggiore, tutto l'attacco richiede 15 minuti di tempo.

2.6 Microarchitectural Data Sampling (MDS Attacks)

Nel 2019-2020 sono state scoperte 4 nuove vulnerabilità che sono state annunciate collettivamente col nome di Microarchitectural Data Sampling [22]; sono tutte vulnerabilità side channel che coinvolgono i processori Intel (nello specifico, i processori di 8° e 9° generazione) e sono basate sul campionamento, sulla raccolta di dati rubati da piccole strutture all'interno della CPU, che variano a seconda dell'attacco. Quest'ultimi sono: RIDL (Rogue In-Flight Data Load) [23], Fallout [24], ZombieLoad [25] e CacheOut [26]. Di seguito viene discusso solo il primo di essi.

2.6.1 RIDL (Rogue In-Flight Data Load)

Premessa

Nelle moderne CPU, ci sono molte fonti potenziali di "dati in volo" (*in-flight data*), come ad esempio il *Re-Order Buffer* (ROB), *Load* e *Store Buffer* (LB e SB) e *Line Fill Buffer* (LFB). Il primo è un buffer che è usato nella gestione dell'esecuzione fuori sequenza, per la precisione nella rinomina dei registri; sostanzialmente mantiene l'ordine originale delle μ-op. Verranno presentati brevemente gli store buffer e i line fill buffer.

Store Buffer Sono buffer interni usati per tenere traccia di salvataggi pendenti e dati in-flight coinvolti in ottimizzazioni come *store-to-load forwarding*[11]. Alcuni processori moderni applicano un forte ordinamento della memoria, dove istruzioni load e store che si riferiscono allo stesso indirizzo fisico non possono essere eseguite fuori sequenza. Tuttavia, siccome la traduzione degli indirizzi è un processo lento, l'indirizzo fisico potrebbe non essere ancora disponibile, e il processore effettua disambiguazione della memoria (insieme di tecniche usate nell'esecuzione fuori ordine, per identificare dipendenze tra le operazioni di memoria - load e store appunto) per predire se le istruzioni di load e store si riferiscono allo stesso indirizzo fisico. Questo permette al processore di eseguire speculativamente istruzioni fuori sequenza di caricamento e salvataggio in modo non ambiguo. Come micro-ottimizzazione, se le istruzioni load e store sono ambigue, il processore può speculativamente effettuare una *store-to-load* forward sui dati dallo store buffer al load buffer.

[11] Supponiamo di effettuare un'operazione di store seguita da un'operazione di read, allo stesso indirizzo. Se i dati non sono ancora stati scritti in memoria, ma si trovano in qualche buffer in attesa di ricevere un commit, rischiamo di leggere un valore sbagliato (quello vecchio); lo store-to-load forwarding permette invece di servire le richieste dal buffer degli store (se sono presenti dati per l'indirizzo richiesto) anziché dalla memoria centrale/cache.

Line Fill Buffer Questi buffer sono buffer interni che la CPU usa per tenere traccia delle richieste di memoria in sospeso, ed effettuano una serie di ottimizzazioni come unire store in-flight multipli. A volte, i dati possono essere già disponibili nel LFB e, come micro-ottimizzazione, la CPU può caricare speculativamente questi dati (ottimizzazioni simili sono eseguite anche su e.g. store buffer). In entrambi i casi, le CPU moderne che implementano esecuzioni speculative aggressive possono speculare senza alcuna consapevolezza degli indirizzi fisici o virtuali coinvolti. RIDL si concentrerà particolarmente su questi buffer.

Attacco RIDL è una classe di attacchi; in questi attachi, l'obiettivo è rubare dati confidenziali da una vittima, come chiavi private e password, abusando le vulnerabilità dell'esecuzione speculative e i buffer nelle CPU Intel. Si assume di attaccare un sistema basato su Intel, con anche le ultime patch di sicurezza installate contro gli attacchi basati sull'esecuzione speculativa. Si assume anche di poter eseguire codice non privilegiato sul sistema della vittima (e.g. sandbox JavaScript, VM o processi utente).

Siccome le fonti del leak possono essere multiple (i.e. ci sono più buffer da cui si possono estrapolare informazioni), considereremo una variante dell'attacco basata sul LFB.

Per quanto riguarda l'attacco, anzitutto il codice della vittima da un altro dominio di sicurezza (e.g. kernel) effettua una load o una store di qualche dato segreto. Internamente, la CPU esegue il load o lo store attraverso qualche buffer interno, come ad esempio il LFB. Successivamente, quando anche l'attaccante esegue una load, il processore usa speculativamente i dati in-flight dal LFB (senza restrizioni sull'indirizzamento) piuttosto che su dati validi. Infine, usando i dati caricati speculativamente come indice in un buffer Flush+Reload (o qualsiasi altro canale nascosto), l'attaccante può estrarre il valore segreto. L'attacco è rappresentato dal seguente frammento di codice (da [23]):

```
1 /* Flush flush & reload buffer entries */
```

```
2  for (k = 0; k < 256; ++k)
3          flush(buffer + k * 1024)
4
5  /* Speculatively load the secret */
6  char value = *(new_page);
7  /* Calculate the corresponding entry */
8  char *entry_ptr = buffer + (1024 * value);
9  /* Load that entry into the cache */
10 *(entry_ptr)
11
12 /* Time the reload of each buffer entry to see which entry is now ca
13 for (k = 0; k < 256; ++k){
14   t0 = cycles();
15   *(buffer + 1024 * k);
16   dt = cycles() - t0;
17
18   if (dt < 100)
19         ++results[k];
20 }
```

Le linee 2-3 svuotano il buffer che sarà poi usato nel canale nascosto per rubare il segreto acceduto speculativamente alla linea 6. Nello specifico, quando si esegue la linea 6, la CPU carica speculativamente dalla memoria nella speranza che sia la nuova pagina allocata dall'attaccante, mentre in realtà è un qualche dato in-flight dal LFB appartenente ad un diverso dominio di sicurezza arbitrario. Quando il processore individua eventualmente il caricamento speculativo scorretto, scarterà tutte le modifiche ai registri o alla memoria, e ricomincerà l'esecuzione dalla linea 6 con il valore corretto. Tuttavia, siccome esisteranno ancora tracce dell'esecuzione speculativa al livello microarchitetturale (nella forma della linea di cache corrispondente), è possibile osservare i dati in-flight rubati usando un semplice canale nascosto Flush+Reload, non diverso da quello di altri attacchi speculativi. Infatti,

il resto del codice riguarda proprio il canale speculativo: le linee 8-10 accedono speculativamente una delle voci nel buffer, usando il dato in-flight rubato come indice. Le linee 12-21 poi accedono tutte le voci nel buffer per vedere se qualcuna di loro è significativamente più veloce (indicando che la linea di cache è presente). Nello specifico, ci si aspetta che *due* accessi siano veloci, non solo quello corrispondente all'informazione sottratta: quando il processore scopre il proprio errore e riparte dalla linea 6 con il valore giusto, il programma accederà il buffer anche con questo indice.

Anche se il concetto base dietro i dati in-flight può essere intuitivo, implementare con successo un attacco si è rivelato difficoltoso [23]: a differenza dei lavori precedenti, costruiti su funzionalità ben documentate come la predizione dei salti, il comportamento di buffer interni della CPU come LFB sono per la maggior parte sconosciuti. Per montare l'attacco, è stato necessario infatti sottoporre la CPU a *reverse engineering*, per comprendere il funzionamento di questi buffer e la loro interazione con la pipeline del processore.

Implementazione Attraverso tecniche di reverse engineering, i ricercatori hanno prima dimostrato per via sperimentale che l'attacco descritto ruba i dati attraverso il LFB e non con altri elementi microarchitetturali. Successivamente, bisogna risolvere il problema della sincronizzazione: ovvero l'attaccante deve far sì che i dati giusti siano nel LFB al momento giusto, sincronizzandosi con la vittima. Ci sono diversi modi per ottenere ciò, le seguenti sono due primitive di esempio:

- **Serializzazione**: le CPU Intel implementano diverse barriere per effettuare vari tipi di serializzazione: `lfence` garantisce che tutte le istruzioni di load lanciate prima di `lfence` diventino globalmente visibili, `sfence` garantisce che lo stesso per le istruzioni store e `mfence` garantisce che sia le load che le store prima di `mfence` diventino globalmente visibili. Per far rispettare questo comportamento, il processore attende che le load e store siano ritirate "prosciugando" i buffer load e/o store e le corrispondenti voci nel LFB. L'istruzione `mfence` quindi forma

un punto di sincronizzazione che permette l'attaccante di osservare le ultime load e store prima che i buffer siano completamente svuotati.

- **Contesa**: un altro modo di sincronizzare vittima e attaccante è creare una contesa con il LFB, costringendo le voci ad essere eliminate. Così facendo è possibile ottenere un po' di controllo sulle voci da rubare.

In un'implementazione su un sistema reale, ci sono delle sfide addizionali che vengono poste. Il requisito fondamentale naturalmente rimane quello di avere dati e.g. con privilegi root in-flight, cosa che si può fare in modo triviale (invocando un binario con permesso `setuid`). Siccome però il LFB è usato in molti contesti, bisogna trovare un modo per ottenere solo i dati di interesse per l'attacco (e filtrare gli altri). Vediamo un attacco pratico dove viene usata una di queste tecniche (allineamento ai dati) e un processo vittima e uno attaccante.

Supponiamo di voler leggere il file `/etc/shadow`, non accessibile ad un normale utente. La prima linea del file contiene la voce per l'utente root; possiamo chiamare continuamente il programma `passwd` da un utente non privilegiato; il processo privilegiato aprirà e leggerà il file `/etc/shadow`, altrimenti inaccessibile per l'attaccante. Possiamo eseguire il programma visto sopra per leggere i dati dal LFB. Per filtrare l'hash di nostro interesse, una tecnica possibile è *l'allineamento ai dati*: conoscendo una parte dei dati da rubare (nel nostro caso l'intestazione `root:`nella prima riga del file), è possibile mascherare i dati che ancora non si conoscono; sottraendo poi il valore noto, i valori che non sono consistenti con quelli precedentemente osservati saranno fuori limite nel buffer FLush+Reload, e quindi non saranno rubati.

In altre parole, si fa un processo di pattern matching; dato un prefisso noto, se dopo una load c'è un match allora possiamo scoprire un nuovo byte rilevante (e modificare il nostro prefisso), se invece non c'è un match i byte non sono rilevanti e si possono scartare (cfr. figura 2.2).

Il primo tentativo con questo attacco ha permesso di recuperare in 24 ore 26 caratteri, lasciando 8 caratteri da rubare. È possibile tuttavia migliorare significativamente la velocità [23].

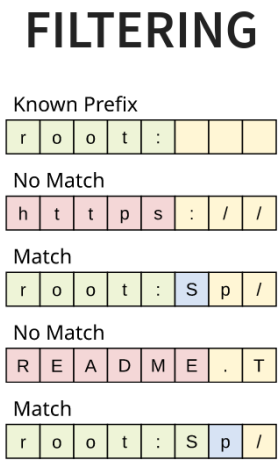

Figura 2.2: Una possibile tecnica in un attacco RIDL per filtrare i dati dai LFB.

Sono stati dimostrati anche diversi altri attacchi (e.g. tra macchine virtuali, al kernel, con JavaScript).

Mitigazioni Intel ha rilasciato un aggiornamento per il microcodice in risposta alla scoperta di RIDL; questo include l'istruzione `verw` che sostanzialmente sovrascrive i buffer.

Un'altra soluzione è quella di disabilitare SMT (simultaneous multithreading), in quanto le informazioni sensibili possono essere rubate dai thread hardware (cfr hyperthreading®), con una sostanziale perdita di performance [23].

2.7 Contromisure per gli attacchi microarchitetturali

Le contromisure proposte finora sono state specifiche per ogni vulnerabilità (e.g. si vedano quelle proposte per RowHammer in 2.5.1).

In ogni caso, sono tutte principalmente contromisure software; qualche contromisura a livello hardware è disponibile nei processori di nuova generazione, ma sembra non essere efficace [28].

Un'importante distinzione che va fatta è tra le contromisure per gli attacchi pre-MDS (e.g. Meltdown, Spectre) e gli attacchi MDS: siccome infatti i primi sono tutti basati sull'abuso dello spazio di indirizzamento, è possibile modificare quest'ultimo per mitigare gli attacchi (nel caso di Meltdown, ad esempio, basta avere due spazi di indirizzamento diversi, uno solo per l'utente e uno per il kernel - che conterrebbe sia indirizzi per il kernel che per lo spazio utente [29]; cfr figura 2.3), usando quindi una tattica generica che potrebbe risolvere il problema più in generale.

Gli attacchi della classe RIDL però non sono basati sull'indirizzamento, di conseguenza tutte le mitigazioni precedenti non sono applicabili, e attualmente non ci sono soluzioni definitive, ma solo dei "palliativi" (come la disabilitazione di SMT).

Molti attacchi inoltre sono stati rivelati in grado di superare alcune delle barriere ritenute più sicure, come la tecnologia intel SGX (Software Guard Extensions [30]), tecnologia che servirebbe ad eseguire codice in un ambiente affidabile (una enclave).

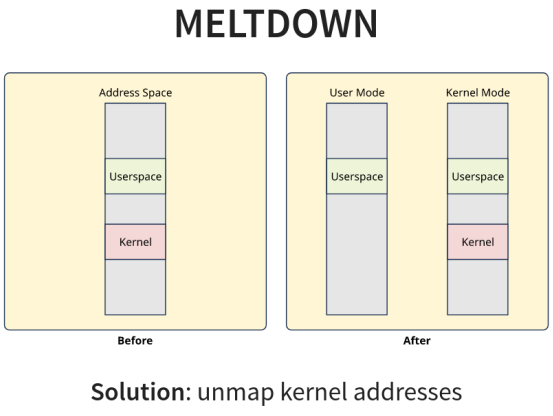

Figura 2.3: Mitigazione di Meltdown.

Verrà discusso però in modo più approfondito una delle mitigazioni principali proposte per Meltdown, KAISER.

2.7.1 KAISER (Kernel Address Isolation to have Side-channels Efficiently Removed)

Fin dall'inizio, Linux ha mappato la memoria del kernel nello spazio degli indirizzi di ogni processo in esecuzione, per motivi di performance e di fiducia nei confronti della MMU (avendo lo scopo di impedire allo spazio utente di accedere allo spazio del kernel).

In un sistema a 32 bit, il layout dello spazio degli indirizzi di un processo in esecuzione è destinato nei primi 3GB (indirizzi da `0x00000000` a `0xbfffffff`) all'utilizzo nello spazio utente, e 1GB finale (da `0xc0000000` a `0x7fffffff`) per il kernel. Ogni processo può vedere la propria memoria nei 3GB in basso, mentre il mapping dello spazio del kernel è lo stesso per tutti.

Nei sistemi contemporanei x86_64 la situazione è analoga. Anche se dallo spazio utente si può vedere lo spazio di indirizzi riservato per il kernel, non vi si può accedere; non vi sono però costrizioni sulla quantità di memoria virtuale che può essere indirizzata. Viene anche applicata un'importante tecnica detta **KASLR**, Kernel Address Space Layout Randomization (derivata di ASLR), che permette di randomizzare la posizione del kernel nello spazio degli indirizzi virtuali durante l'avvio del sistema. L'idea è quella di impedire all'attaccante di sapere dove il kernel risiede nella memoria, rendendo molto più difficili diversi tipi di attacchi. Siccome però è stato dimostrato che KASLR si può aggirare, una risposta a questa problematica risiede in KAISER.

KAISER è un'implementazione per separare gli spazi di indirizzi per i kernel per sistemi x86_64. Mentre i sistemi attuali hanno un singolo insieme di page table per ogni processo, KAISER ne implementa 2: il primo insieme è sostanzialmente inalterato, ed include sia gli indirizzi del kernel-space che dello user-space, ma è usato solamente quando il sistema è in modalità kernel. La seconda tabella delle pagine, detta "shadow", contiene una copia di tutte

le corrispondenze (mapping) degli indirizzi dello spazio utente, senza lo spazio kernel. C'è invece un insieme minimale sul mapping dello spazio kernel che forniscono le informazioni necessarie per gestire interrupt e system call.

Quando un processo lavora in modalità utente, la tabella shadow sarà attiva. Quando il sistema deve passare nella modalità kernel, verrà eseguito uno switch all'altro set di tabelle.

La difesa comunque non è completa, e richiede un costo addizionale in termini di performance.

2.7.2 Identificazione di un attacco in corso

La via alternativa alla prevenzione è la *detection*, l'identificazione di un attacco in corso. Alla base di questo metodo c'è l'osservazione dei comportamenti anomali dei processi in un sistema; questo però pone non pochi problemi, in quanto gli attacchi microarchitetturali sono ancora in una fase di studio; di conseguenza, non ci sono sufficienti modelli per permettere un'analisi precisa di quali siano esattamente i comportamenti anomali. Bisogna anche citare il fatto che non ci sia ancora una classificazione ed una tassonomia comune [23], il che sottolinea ulteriormente come questo sia un campo ancora in una fase infantile, e ci sia quindi molto da ricercare.

Infine, in mancanza di modelli precisi, non è possibile prevenire nemmeno futuri attacchi, ma solo creare patch per le varianti attualmente rivelate pubblicamente.

L'identificazione degli attacchi basata su anomalie cerca di risolvere le sfide citate sopra modellando il comportamento delle applicazioni "benigne" e cercando tutte quelle che fuoriescono dai modelli definiti. I ricercatori hanno quindi proposto soluzioni come *FortuneTeller* [32], un sistema basato sul fingerprint di queste applicazioni benigne; FortuneTeller è il primo modello/tecnica generico per identificare attacchi microarchitetturali. Esso si basa sul Deep Learning: infatti, altre tecniche come metodi statistici non sono

adatti a risolvere il problema, a causa dell'enorme complessità delle moderne microarchitetture [32].

FortuneTeller inoltre si basa sui *performance hardware counters*, che sono già disponibili nei processori attualmente in commercio, il che significa che non richiede hardware addizionale, ma può essere applicato subito. FortuneTeller è stato in grado di identificare sia attacchi alla cache (Flush+Flush), sia attacchi basati su esecuzioni transienti (Meltdown, Spectre) sia RowHammer.

2.8 Discussione finale e conclusione

Nei capitoli precedenti abbiamo introdotto gli attacchi microarchitetturali automatizzabili via software, una nuova classe di attacchi che non dipendono dal sistema operativo installato o più in generale da bug software, ma solamente dall'architettura sottostante. Questi attacchi sono inoltre in grado di bypassare sistemi SandBox e altre soluzioni ritenute tradizionalmente sicure come il mapping di indirizzi del kernel nello spazio utente.

Abbiamo visto che esistono soluzioni software, che però funzionano (in parte) solo per prevenire specifici attacchi, e non ci sono ancora soluzioni generiche adottate in modo omogeneo dai produttori di sistemi operativi e di software più in generale.

Molte di queste patch inoltre sono difficili da applicare, in quanto richiederebbero la ricompilazione dei programmi per introdurre nuove istruzioni assembly, cosa quasi impossibile per il software legacy.

Infine, per quanto riguarda le soluzioni hardware, ovviamente esse possono essere applicate unicamente nei nuovi processori, e non nei vecchi, che continuerebbero quindi a funzionare e a fare affidamento solo a protezioni via software. Questo inoltre sottolinea come ci sia la necessità di piattaforme hardware open source, come ad esempio RISC-V[12]: molti di questi bug erano infatti presenti da tempo, ma soltanto in questi ultimi anni i ricercatori sono riusciti a capire il funzionamento degli elementi microarchitetturali non do-

[12] `https://riscv.org/`

cumentati necessari per studiare le vulnerabilità poi pubblicate; il fatto che un ricercatore collabori direttamente con le aziende coinvolte per segnalare il problema, non elimina la possibilità che sulla base del lavoro attuale futuri attori con cattive intenzioni possano abusare di queste scoperte per scopi personali, attaccando sistemi esistenti. Architetture open source permetterebbero invece di identificare molto prima i problemi, in quanto lo sforzo per capirne il funzionamento sarebbe molto più esiguo essendo tutto documentabile, e non dovendo usare processi di reverse engineering.

Cercheremo però nella prossima sezione di discutere la gravità della minaccia.

2.8.1 Pericolosità della minaccia

Le vulnerabilità fin qui esposte fanno intendere che tutte le nostre informazioni personali e i nostri dati sensibili sono attualmente "allo scoperto", siano esse salvate nel nostro computer, smartphone o nel cloud. Ma è davvero così drammatica la situazione?

Non ci sono ancora dati sufficienti per stabilire se siano già avvenuti attacchi usando queste tecniche, e fornire quindi ragionamenti conclusivi basati su dati reali. È possibile però fare una serie di congetture, volgendo lo sguardo alle esperienze passate.

Anzitutto, non è detto che ci sia un reale motivo per cui un malintenzionato debba decidere di usare queste specifiche tecniche: se prendiamo il caso di WannaCry, esso è un ransomware che nel 2017 ha colpito un numero di computer dell'ordine delle migliaia in tutto il mondo, per il quale le patch di sicurezza erano già presenti, ma non erano state applicate dagli utenti per svariati motivi; essendo questi attacchi apparentemente ancora relegati al mondo accademico, è quindi verosimile ipotizzare che un avversario decida di usare tecniche più "collaudate", come appunto ransomware, trojan, backdoor, keylogger, MITM etc.

Questi attacchi sono più interessanti se visti rispetto al mondo del cloud: infatti, avendo dimostrato che è possibile aggirare meccanismi di virtualizza-

zione come Docker, un avversario potrebbe essere interessato a rubare quanti più dati possibili da un'infrastruttura del genere, siccome l'utilizzo di sistemi cloud è anche notevolmente in espansione. Tuttavia, uno dei vantaggi del cloud è proprio quello di avere un'entità, diversa dall'utente finale, che si occupi della sicurezza dell'infrastruttura e di conseguenza cerchi di tenerla aggiornata e protetta, anche proprio da queste nuove minacce. Bisogna inoltre sottolineare che alcuni di questi attacchi sono mirati a rubare informazioni segrete da specifiche applicazioni target, e non a sottrarre informazioni in modo indiscriminato come potrebbe fare un keylogger, il che suggerirebbe che gli attacchi potrebbero essere rivolti solo a specifiche persone (che magari sanno già di essere dei bersagli) o a specifiche informazioni, e non generici e indiscriminati contro qualsiasi utente.

Per quel che riguarda invece la sicurezza dei computer e dei telefoni, è degno di nota la possibilità di poter applicare questi attacchi attraverso JavaScript o applicazioni per smartphone; ma non è certo una novità l'esistenza di applicazioni e siti malevoli, per i quali si possono applicare le solite *good practice* relative alla sicurezza, come ad esempio scaricare e installare applicazioni solo da fonti attendibili, visitare siti ritenuti sicuri etc.

In conclusione, per quanto gli attacchi microarchitetturali pongano nuove sfide per la sicurezza e siano sicuramente potenzialmente molto pericolosi data la scarsità di soluzioni, la vastità di scenari e piattaforme hardware in cui si possono applicare e la loro ancora scarsa conoscenza, non ritengo personalmente che siano in grado di sovvertire l'ordine naturale delle cose come potrebbe essere invece e.g. la dimostrazione che P = NP per quanto riguarda la crittografia; non sono certamente da sottovalutare (è stato appunto citato l'esempio di WannaCry, un ransomware per il quale le patch esistevano già), ma non ritengo che pongano minacce incredibilmente più pericolose rispetto a quelle già esistenti.

Appendice A

Algoritmo Binario di esponenziazione SM (Square and Multiply)

RSA è il sistema crittografico a chiave pubblica più usato [9]. La principale computazione nelle decriptazioni/criptazioni è l'esponenziazione modulare $P = M^d(modN)$, dove M è il messaggio o testo cifrato, d è la chiave privata e N è il modulo pubblico. N qui è il prodotto di due numeri primi p e q. Siccome la dimensione della chiave è molto grande, e.g. 1024 bit, l'esponenziazione è molto costosa in termini di tempo d'esecuzione. Quindi, implementazioni effettive di RSA usano algoritmi efficienti per calcolare il risultato di questa operazione.

Il problema dell'esponenziazione è formulato nel modo seguente [11]: dati $a, n \in \mathbb{N}$ si vuole calcolare l'intero $c = a^n$.

Indicando con t il numero di cifre binarie necessarie per rappresentare l'intero n, cioè:

$$t = \lceil \log_2 \rceil, n = (n_{t-1}, ..., n_1, n_0) \text{ con } n_i \in \{0, 1\}, i \in \{0, 1, ..., t-1\}$$

possiamo scrivere:

$$c = a^n = a^{\sum_{j=0}^{t-1} n_j 2^j} = a^{n_{t-1}+n_{t-2}2^{t-2}+...+n_1 2^1+n_0}$$

A seconda del modo in cui è possibile leggere l'ultimo membro della catena di uguaglianze appena scritte si esibiscono diverse formalizzazioni dell'algoritmo noto come Square and Multiply (i.e. scansione verso destra, oppure scasione da destra a sinistra).

La versione binaria di SM è il modo più semplice per fare un'esponenziazione [9]. Vogliamo quindi calcolare $M^d(modN)$, dove d è un numero ad n bit: $d = (d_0, d_1, ..., d_{n-1})$. Nel codice sottostante possiamo vedere un algoritmo per SM binario che scandisce i bit di d da sinistra a destra. Tutte le moltiplicazioni e gli elevamenti a quadrato sono mostrati come operazioni modulari, nonostante l'algoritmo SM di base calcoli esponenziazioni regolari.

```
S = M
for      i from 1 to n - 1 do
                  S = S * S (mod N)
                  if d_i = 1 then
                     S = S * M (mod N)
return S
```

Sapendo quale sarà la direzione del salto condizionale, possiamo determinare i bit d_i della chiave. Anche non conoscendo tutti i bit d_i, occorre scoprirne un numero sufficiente per ricreare tutta la chiave d.

Appendice B

RSA

Il primo crittosistema a chiave pubblica è dovuto a Rivest, Shannon e Adleman. La sicurezza del sistema è dovuta alla difficoltà di fattorizzare in modo efficiente numeri interi della forma $N = p \cdot q$, con n, p primi.

Generazione delle chiavi Essendo un cifrario asimmetrico, esistono due chiavi, una pubblica e una privata. I passi da seguire sono i seguenti:

1. genera due numeri primi p, q di dimensione opportuna (e.g. ≥ 1024 bit);
2. calcola $N = p \cdot q$ e $\phi(N) = (p-1)(q-1)$;
3. scegli un valore e casuale, con $1 < e < \phi(N)$ e $\text{MCD}(e, \phi(N)) = 1$ (i due valori devono essere primi tra loro);
4. calcola $d \equiv e^{-1} (\text{mod} \phi(N))$.

La chiave privata *sk* sarà data da d, mentre la chiave pubblica *pk* sarà data dalla coppia (N, e).

Cifratura e decifratura Dato un messaggio *pk* e una chiave pubblica *pk = (N, e)*, il messaggio cifrato c è dato da:

$$c = \text{RSA}_{pk}(m) = m^e \text{mod } N$$

Dato un testo cifrato c e la chiave segreta $sk = d$, il messaggio in chiaro m è:

$$m = \mathrm{RSA}_{sk}^{-1}(c) = c^d \mathrm{mod}\ N$$

Bibliografia

[1] Daniel Gruss. Microarchitectural Attacks and Beyond. Graz University of Technology

[2] Andrew S. Tanenbaum, Todd Austin. Architettura dei calcolatori Un approccio strutturale, Pearson Italia, 2013.

[3] Ben Lee. Dynamic Branch Prediction. Electrical and Computer Engineering, Oregon State University. `http://web.engr.oregonstate.edu/~benl/Projects/branch_pred/`

[4] `https://web.njit.edu/~rlopes/Mod5.3.pdf`

[5] P. C. Kocher. Timing Attacks on Implementations of Diffe- Hellman, RSA, DSS, and Other Systems. In: Crypto?96. 1996 (pp. 5, 6, 30).

[6] Daniel Gruss. Software-based Microarchitectural Attacks. p. 30

[7] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. (Extended Version)

[8] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack.

[9] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting Secret Keys Via Branch Prediction.

[10] Dmitry Evtyushkin et al. BranchScope: A New Side-Channel Attack on Directional Branch Predictor.

[11] Gerardo Pelosi. Appunti di Crittografia. Dipartimento di Elettronica e Informazione, Politecnico di Milano. http://home.deib.polimi.it/pelosi/lib/exe/fetch.php?media=teaching:001_algebra-ii.pdf

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom Independent (www.paulkocher.com), Google Project Zero, G DATA Advanced Analytics, University of Pennsylvania and University of Maryland, Graz University of Technology, Cyberus Technology, Rambus, Cryptography Research Division, University of Adelaide and Data61. Spectre Attacks: Exploiting Speculative Execution.

[13] BPF: the universal in-kernel virtual machine, in https://lwn.net/Articles/599755/

[14] Meltdown: Reading Kernel Memory from User Space. Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, Graz University of Technology, Cyberus Technology GmbH, G-Data Advanced Analytics, Google Project Zero, Independent (www.paulkocher.com), University of Michigan, University of Adelaide & Data61, Rambus, Cryptography Research Division

[15] Cloud computing - statistics on the use by enterprises; https://ec.europa.eu/eurostat/statistics-explained/pdfscache/37043.pdf

[16] AMD Product Security; https://www.amd.com/en/corporate/product-security

[17] Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. Yoongu Kim, Ross Daly, Jeremie Kim Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, Onur Mutlu, Carnegie Mellon University, Intel Labs

[18] RowHammer: A Retrospective. Onur Mutlu, Jeriemie S. Kim, ETH Zurich, Carnegie Mellon University.

[19] DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard, Graz University of Technology in 25th USENIX Security Symposium.

[20] Rowhammer Attacks: An Extended Walkthrough Guide. Daniel Gruss, Graz University of Technology. `https://gruss.cc/files/sba.pdf`

[21] V. van der Veen et al., Drammer: Deterministic Rowhammer Attacks on Mobile Platforms, CCS, 2016.

[22] Side Channel Vulnerability Microarchitectural Data Sampling, `https://www.intel.it/content/www/it/it/architecture-and-technology/mds.html`

[23] RIDL: Rogue In-Flight Data Load, Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida

[24] Fallout: Leaking Data on Meltdown-resistant CPUs, Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, Yuval Yarom

[25] ZombieLoad: Cross-Privilege-Boundary Data Sampling, Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, Daniel Gruss

[26] CacheOut: Leaking Data on Intel CPUs via Cache Evictions, van Schaik, Stephan and Minkin, Marina and Kwong, Andrew and Genkin, Daniel and Yarom, Yuval

[27] Crittografia nel Paese delle Meraviglie. Daniele Venturi, Springer, 2012; pp. 152-153

[28] https://mdsattacks.com/

[29] RIDL: Rogue In Flight Data Load; Stephan van Schaik. Presented at the 2019 IEEE Symposium on Security & Privacy. https://www.youtube.com/watch?v=1Y0h4JyK3fs&t=797s

[30] Enhanced Security Features for Applications and Data In-use. https://software.intel.com/sites/default/files/managed/c3/8b/intel-sgx-product-brief-2019.pdf

[31] KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/

[32] FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning. Berk Gulmezoglu, Ahmad Moghimi, Thomas Eisenbarth and Berk Sunar.